Real-Time Compute Virtualization

Luca Abeni
luca.abeni@santannapisa.it

March 30, 2020

Real-Time in VMs???

- Running real-time applications on an RTOS is not a problem...
- ...But, can real-time applications run in virtual machines?
    - Real-Time in Virtual Machines??? But... Why?
- Component-Based Development
    - Complex applications: sets of smaller components
    - Both functional and temporal interfaces
- Security (isolate real-time applications in a VM)
- Easy deployment; Time-sensitive clouds

Real-Time in VMs

- Real-Time applications running in a VM?
  - As for OSs, two different aspects

Real-Time in VMs

- Real-Time applications running in a VM?
  - As for OSs, two different aspects
    - Resource allocation/management (scheduling)
  - CPU allocation/scheduling: lot of work in literature

Real-Time in VMs

- Real-Time applications running in a VM?
  - As for OSs, two different aspects
    - Latency (host and guest)
  - Latencies not investigated too much (yet!)

Real-Time in VMs

- Real-Time applications running in a VM?
  - As for OSs, two different aspects
    - Resource allocation/management (scheduling)
    - Latency (host and guest)
  - CPU allocation/scheduling: lot of work in literature
  - Latencies not investigated too much (yet!)
- Virtualization: full hw or OS-level
  - Containers: real-time performance of the host kernel
  - Hw virtualization: hypervisors (example: KVM or Xen) can introduce latencies!

Latency

- Latency: measure of the difference between the theoretical and actual schedule
  - Task τ expects to be scheduled at time t ...
  - ... but is actually scheduled at time t'
  - $\Rightarrow$ Latency $L = t' - t$
- The latency L can be accounted for in schedulability analysis
  - Similar to what is done for shared resources, etc...
  - Strange “shared resource”: the OS kernel (or the hypervisor)

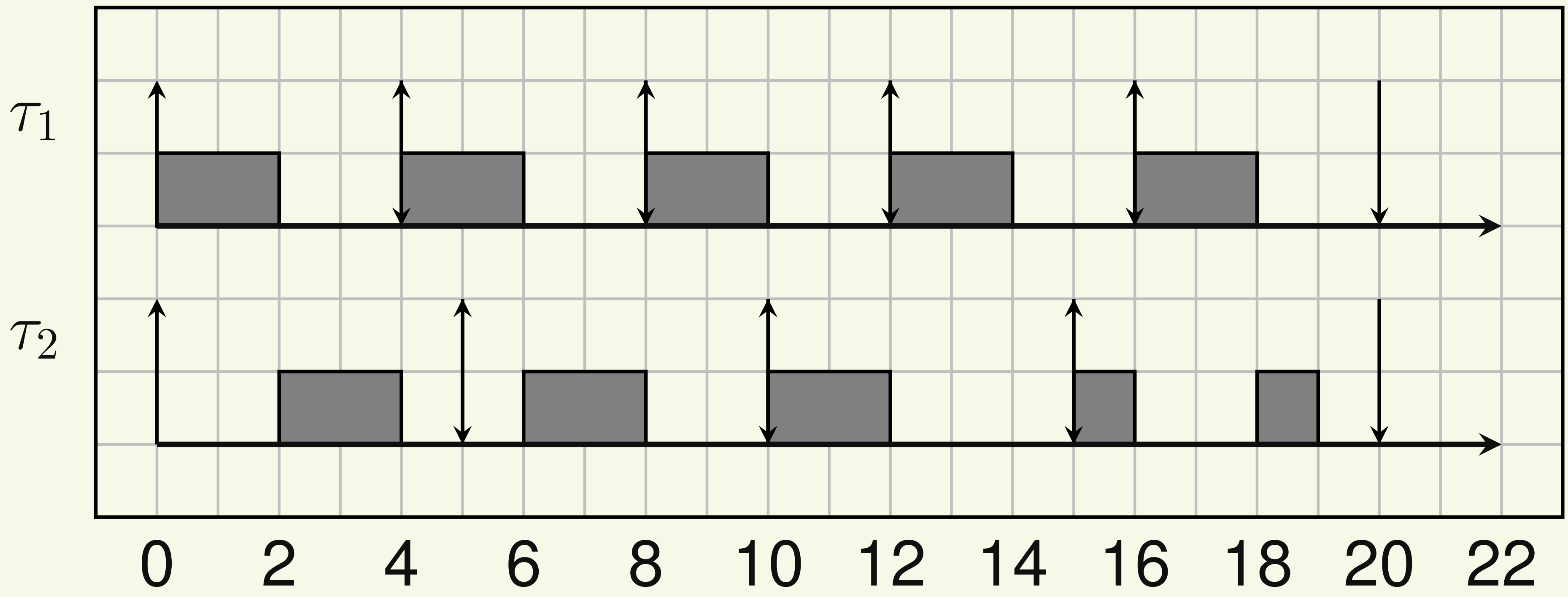
Example: Periodic Task

- Consider a periodic task

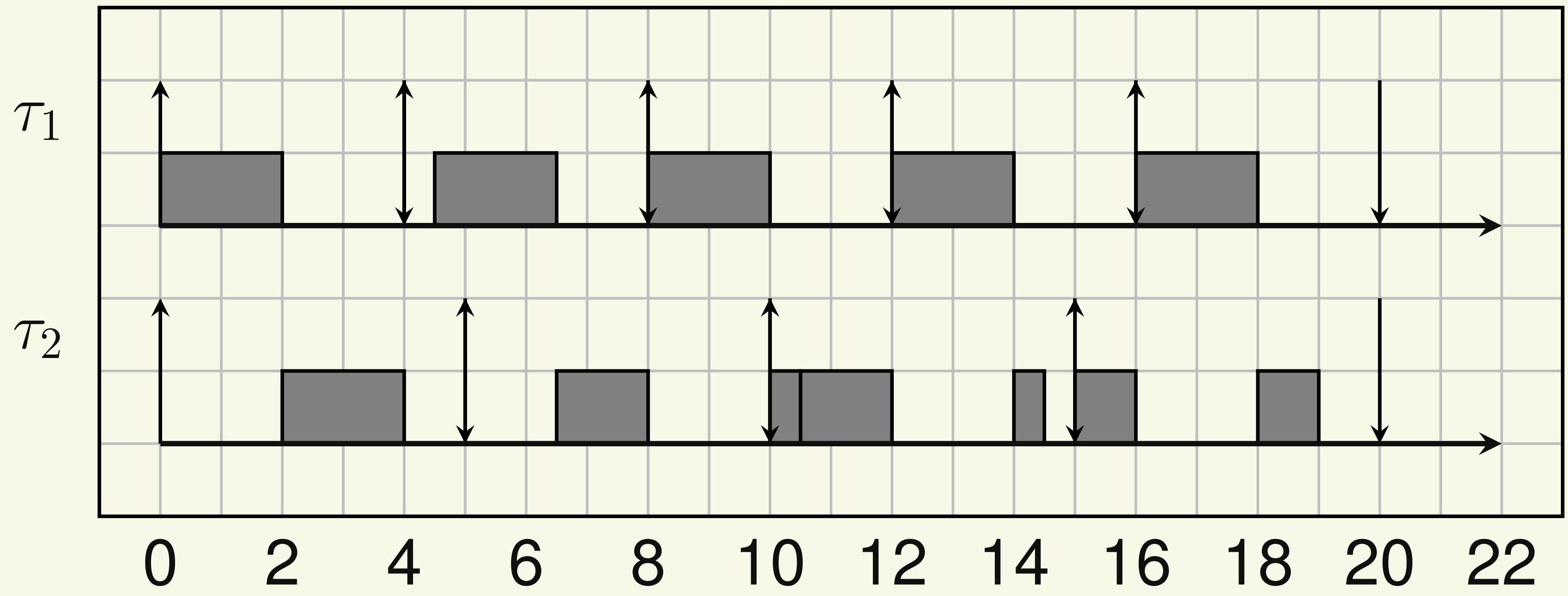
```
/* ... */
while(1) {
  /* Job body */
  clock_nanosleep(CLOCK_REALTIME,
                  TIMER_ABSTIME, &r, NULL);
  timespec_add_us(&r, period);
}
```

- The task expects to be executed at time `r` $(= r_0 + jT)$...
- ...But is sometimes delayed to $r_0 + jT + \delta$

Theoretical Schedule



Actual Schedule



- What happens if the 2^{nd} job of τ_1 arrives a little bit later???
  - The 2^{nd} job of τ_2 misses a deadline!!!

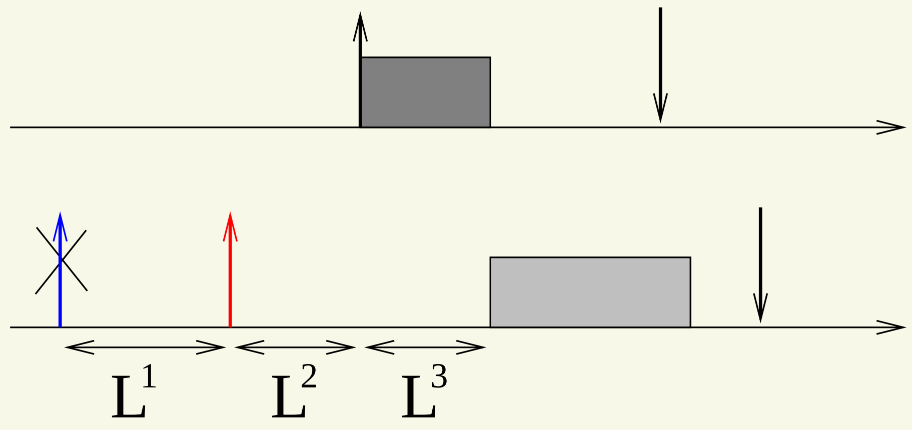
Effects of the Latency

- Upper bound for L? If not known, no schedulability analysis!!!
  - The latency must be *bounded*: $\exists L^{max} : L < L^{max}$
- If L^{max} is too high, only few task sets result to be schedulable
  - The worst-case latency L^{max} cannot be too high

Sources of Latency — 1

- Task: stream of jobs (activations) arriving at time r_j
- Task scheduled at time $t' > r_j \rightarrow$ Delay $t' - r_j$ caused by:

  1. Job arrival (task activation) signaled at time $r_j + L^1$
  2. Event served at time $r_j + L^1 + L^2$
  3. Task actually scheduled at $r_{i,j} + L^1 + L^2 + I$



Sources of Latency — 2

- $L = L^1 + L^2 + I$
- I: interference from higher priority tasks
  - Not really a latency!!!
- L^2: *non-preemptable section latency* L^{np}
  - Due to non-preemptable sections in the kernel (or hypervisor!) or to deferred interrupt processing
- L^1: delayed interrupt generation
  - Generally small
  - Hardware (or virtualized) timer interrupt: *timer resolution latency* L^{timer}

Latency in Linux

- Tool (`cyclictest`) to measure the latency
  - Periodic task scheduled at the highest priority
  - Response time equal to execution time (almost 0)
- Vanilla kernel: depends on the configuration
  - Can be tens of milliseconds
- Preempt-RT patchset (`https://wiki.linuxfoundation.org/realtime`): reduce latency to less than 100 microseconds
  - Tens of microseconds on well-tuned systems!
- So, real-time on Linux is not an issue
  - Is this valid for hypervisors/VMs too?

What About VM Latencies?

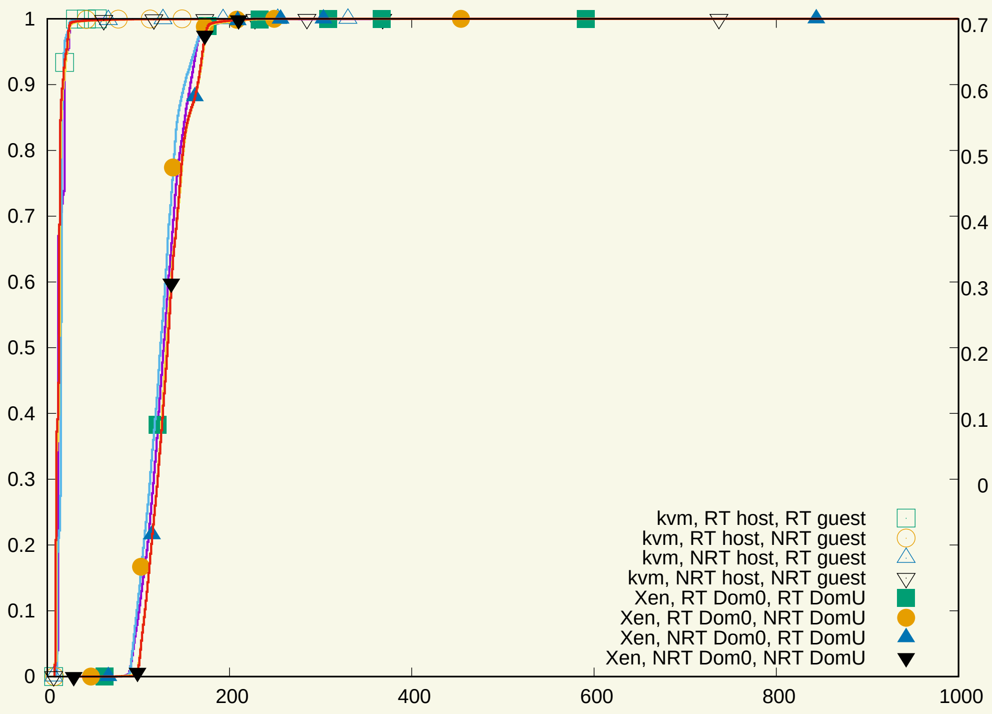
- Hypervisor: software component responsible for executing multiple OSs on the same physical node
  - Can introduce latencies too!
- Different kinds of hypervisors:
  - Xen: bare-metal hypervisor (*below* the Linux kernel)
    - Common idea: the hypervisor is small/simple, so it causes small latencies
  - KVM: hosted hypervisor (Linux kernel module)
    - Latencies reduced by using Preempt-RT
    - Linux developers already did lot of work!!!

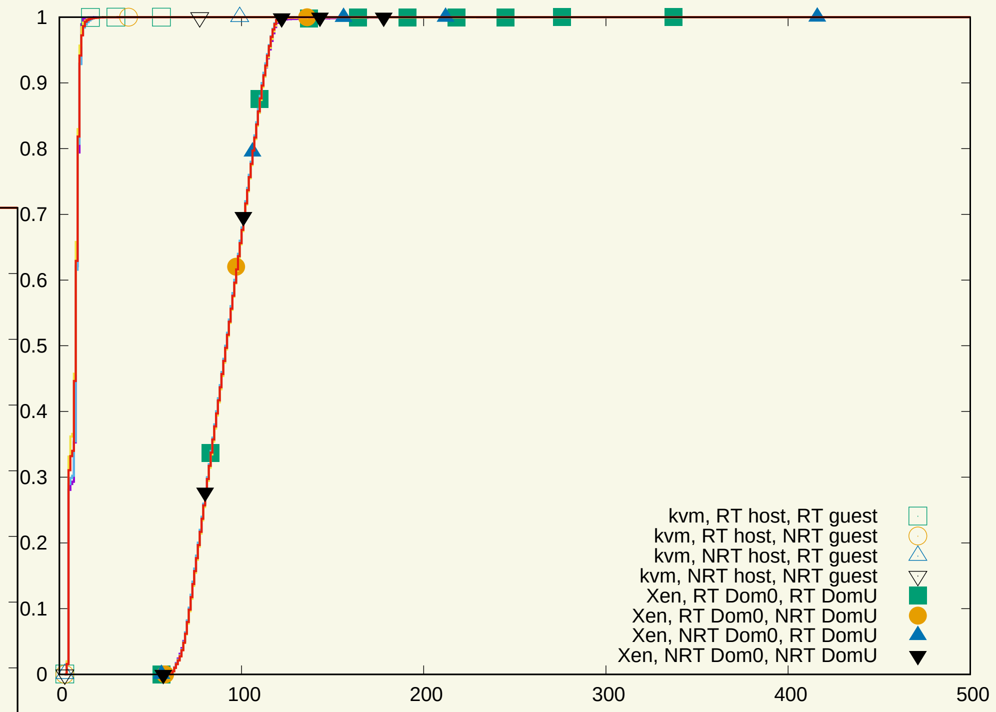
Hypervisor Latency

- Same strategy/tools used for measuring kernel latency
- Idea: run `cyclictest` in a VM
  - `cyclictest` process ran in the guest OS...
  - ...instead of host OS
- `cyclictest` period: $50\mu s$
- “Kernel stress” to trigger high latencies
  - Non-real-time processes performing lot of syscalls or triggering lots of interrupts
  - Executed in the host OS (for KVM) or in Dom0 (for Xen)
- Experiments on multiple x86-based systems

Hypervisor Latencies

Intel Core Duo





Intel Core i7

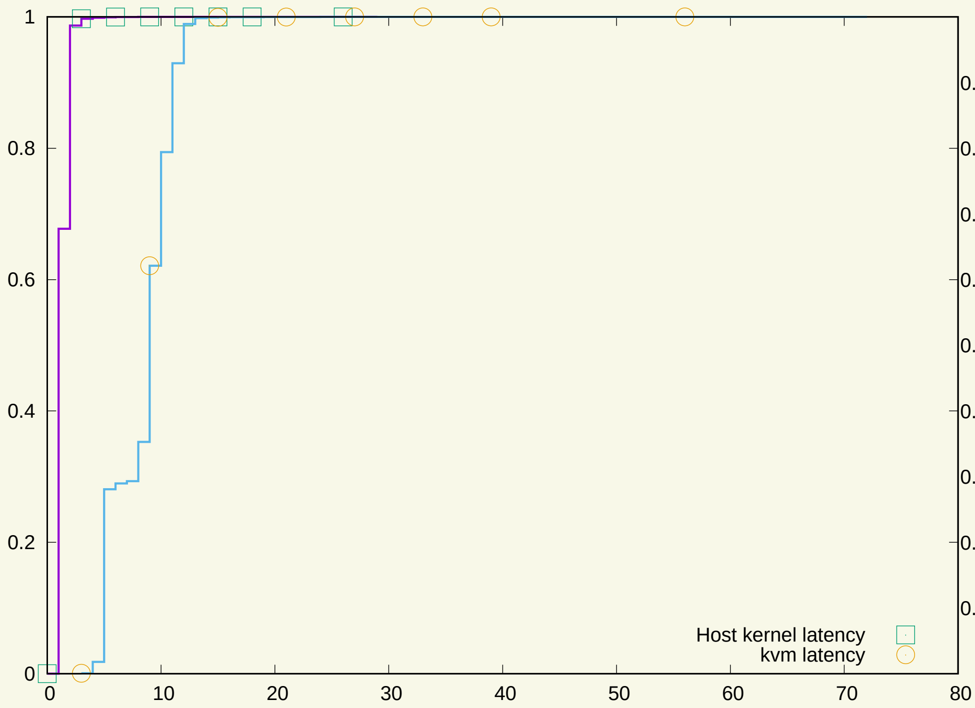
Worst Cases

Kernels	Core Duo		Core i7	
	Xen	KVM	Xen	KVM
NRT/NRT	$3216\mu s$	$851\mu s$	$785\mu s$	$275\mu s$
NRT/RT	$4152\mu s$	$463\mu s$	$1589\mu s$	$243\mu s$
RT/NRT	$3232\mu s$	$233\mu s$	$791\mu s$	$99\mu s$
RT/RT	$3956\mu s$	$71\mu s$	$1541\mu s$	$72\mu s$

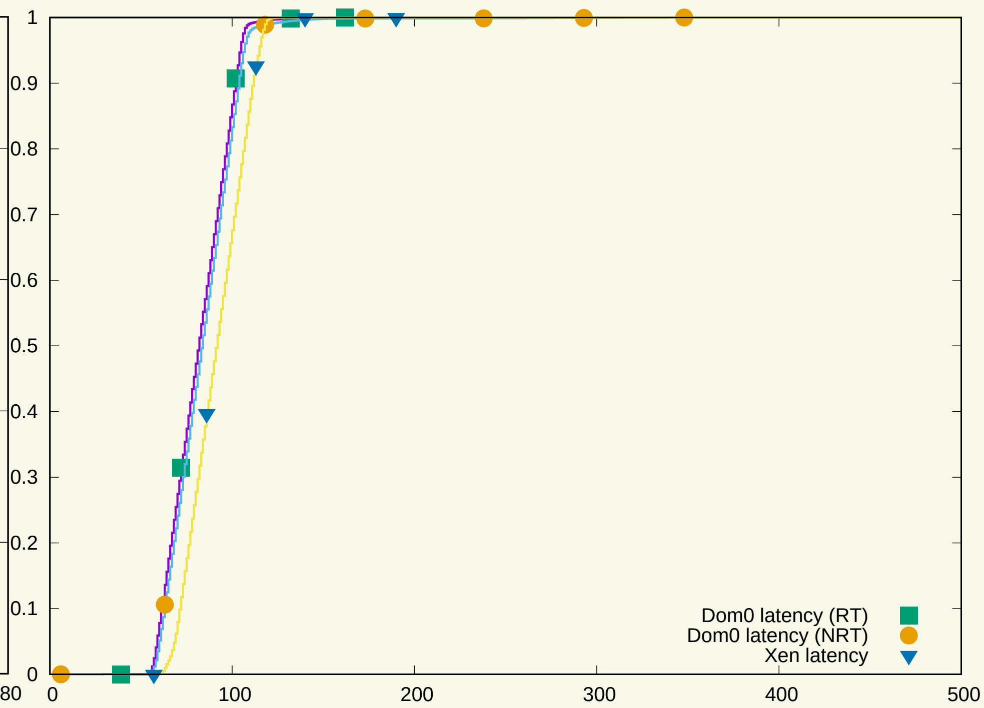
- Preempt-RT helps a lot with KVM
  - Good worst-case values (less than $100\mu s$)
- Preempt-RT in the guest is dangerous for Xen
  - Worst-case values stay high

Hypervisor vs Kernel

KVM



Xen



- Worst Cases:
  - Host: $29\mu s$
  - Dom0: $201\mu s$ with Preempt-RT, $630\mu s$ with NRT

Investigating Xen Latencies

- KVM: usable for real-time workloads
- Xen: strange results
  - Larger latencies in general
  - Using Preempt-RT in the guest increases the latencies?
- Xen latencies are not due to the hypervisor's scheduler
  - Repeating the experiments with the null scheduler did not decrease the experienced latencies

Impact of the Kernel Stress

- Experiments repeated without “Kernel Stress” on Dom0
  - This time, using Preempt-RT in the guest reduces latencies!
  - Strange result: Dom0 load *should not* affect the guest latencies...

Kernels	Core Duo		Core i7	
	Stress	No Stress	Stress	No Stress
NRT/NRT	$3216\mu s$	$3179\mu s$	$785\mu s$	$1607\mu s$
NRT/RT	$4152\mu s$	$1083\mu s$	$1589\mu s$	$787\mu s$
RT/NRT	$3232\mu s$	$3359\mu s$	$791\mu s$	$1523\mu s$
RT/RT	$3956\mu s$	$960\mu s$	$1541\mu s$	$795\mu s$

Virtualization Mechanisms

- Xen virtualization: PV, HVM, PVH, ...
  - PV: everything is para-virtualized
  - HVM: full hardware emulation (through qemu) for devices (some para-virtualized devices, too); use CPU virtualization extensions (Intel VT-x, etc...)
  - PVH: hardware virtualization for the CPU + para-virtualized devices (trade-off between the two)

Guest Kernel	PV	PVH	HVM
NRT	$661\mu s$	$1276\mu s$	$1187\mu s$
RT	$178\mu s$	$216\mu s$	$4470\mu s$

What's up with HVM?

- HVM uses qemu as DM
  - qemu instance running in Dom0
  - Used for boot and emulating some devices...
  - ...But somehow involved in the strange latencies!!!
- Scheduling all qemu threads with priority 99, the worst-case latencies are comparable with PV / PVH!!!
  - High HVM latencies due to the Kernel Stress workload preempting qemu...
- Summing up: for good real-time performance, use PV or PVH!

Cyclictest Period

- Most of the latencies larger than cyclictest period...
- Are hypervisor's timers able to respect that period?
  - Example of timer resolution latency...
- So, let's try a larger period!
  - $500\mu s$ and $1ms$ instead of $50\mu s$
  - Measure timer resolution latency $\rightarrow$ no kernel stress
- Results are much better!
  - $P = 500\mu s$: worst-case latency $112\mu s$ (HVM), $82\mu s$ (PVH) or $101\mu s$ (PV)
  - $P = 1000\mu s$: worst-case latency $129\mu s$ (HVM), $124\mu s$ (PVH) or $113\mu s$ (PV)

Further Analysis

- Xen latencies seem to be mainly due to timer resolution latency
  - Turned out to be an issue in the Linux code handling Xen's para-virtualized timers
    - Linux jargon: “clockevent device”
  - Does not activate a timer at less than $100\mu s$ from current time (`TIMER_SLOP`)
- After reducing the timer slop, average latency smaller than $50\mu s$ even for cyclictest with period $50\mu s$
  - Still larger than KVM latencies (probably due to non-preemptable sections?)

Reproducible Results

- Results can be reproduced on your test machine
  - You just need some manual installation of KVM, Xen, etc...

`http://retis.santannapisa.it/luca/VMLatencies`

- Scripts to reproduce the previous experiments
  - Number depends on the hw, but the obtained figures are consistent with the previous results
- The other figures can be easily ontained modifying scripts / configuration files

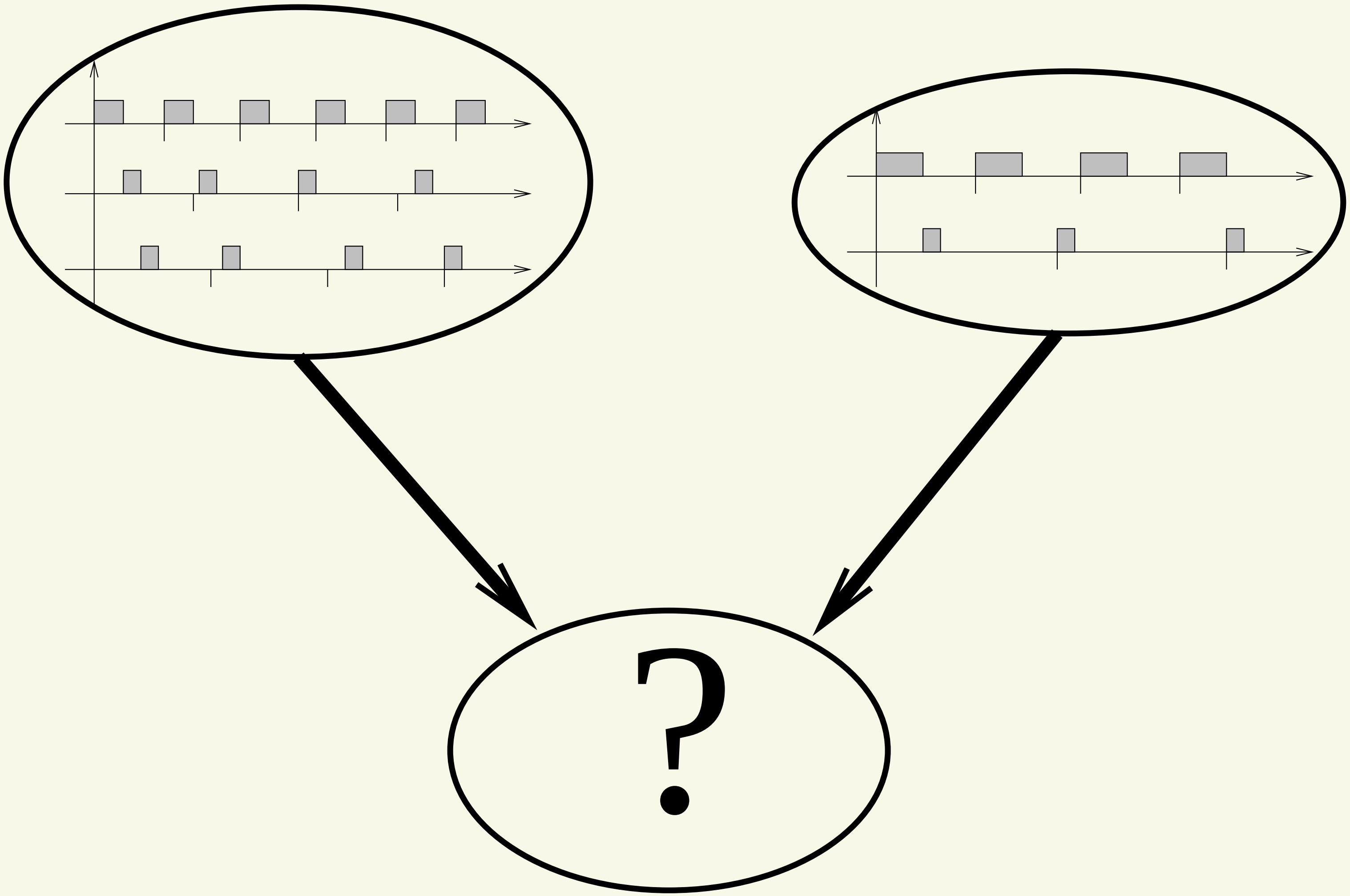
Summing Up

- Latencies experienced in a VM (`cyclictest`)
  - KVM: Preempt-RT allows to achieve low latencies $\rightarrow$ usable for real-time
  - Xen: high latencies, Preempt-RT does not help, strange impact of the Dom0 load
- Xen behaves better when PV or PVH is used
  - Part of the latencies due to the DM (qemu running in Dom0)?
- Xen experiences a large timer resolution latency
  - Fixable by modifying the guest kernel

Latencies and Scheduling

- Most of the industrial work on real-time virtualization focused on latency reduction
  - Example: real-time KVM industrial solution based on vCPU pinning — No scheduling!!!
- Scheduling VMs is still needed to share hardware resources...
  - Bounded latencies are needed to have precisa and accurate vCPU scheduling...
  - ...But appropriate scheduling algorithms are still needed!!!
- Advanced scheduling algoritms are useless if latencies are not bounded, and bounded latencies are useless if appropriate scheduling is not used!

Combining Real-Time Guarantees



- Schedulability analysis in each VM...
- What about the resulting system?

Real-Time Applications Inside VMs

- VM $\mathcal{C}^i$ contains n^i tasks
- How to analyze its schedulability?
  - We only know how to schedule single tasks...
  - And we need to somehow "summarise" the requirements of a VM!

$$\mathcal{C}^i = \{(C_0^i, D_0^i, T_0^i), (C_1^i, D_1^i, T_1^i), \ldots, (C_{n^i}^i, D_{n^i}^i, T_{n^i}^i)\}$$

- So, 2 main issues:

1. Describe the temporal requirements of a VM in a simple way
2. Schedule the VMs, and somehow "combine" their temporal guarantees

The “not so smart” Solution

- Each VM is a set of real-time tasks:

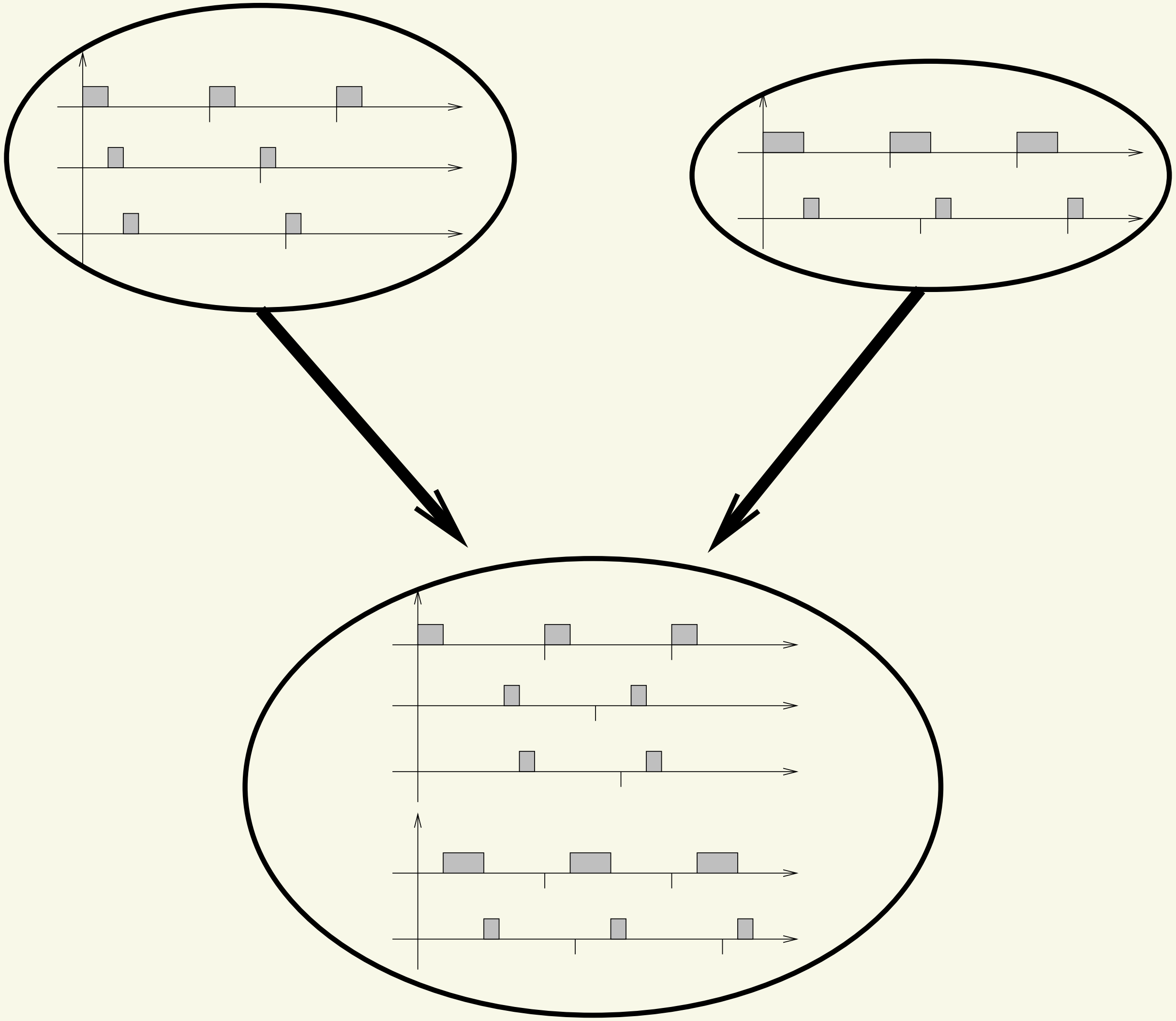
$$\mathcal{C}^i = \{(C^i_j, D^i_j, T^i_j)\}$$

- Build the “global taskset” composed by all the tasks from all the VMs

$$\Gamma = \bigcup_i \mathcal{C}^i$$

- ...And use some known real-time scheduler (RM, EDF, ...) on Γ!

Flattened Scheduling



- One single “flattened” scheduler seeing all the tasks

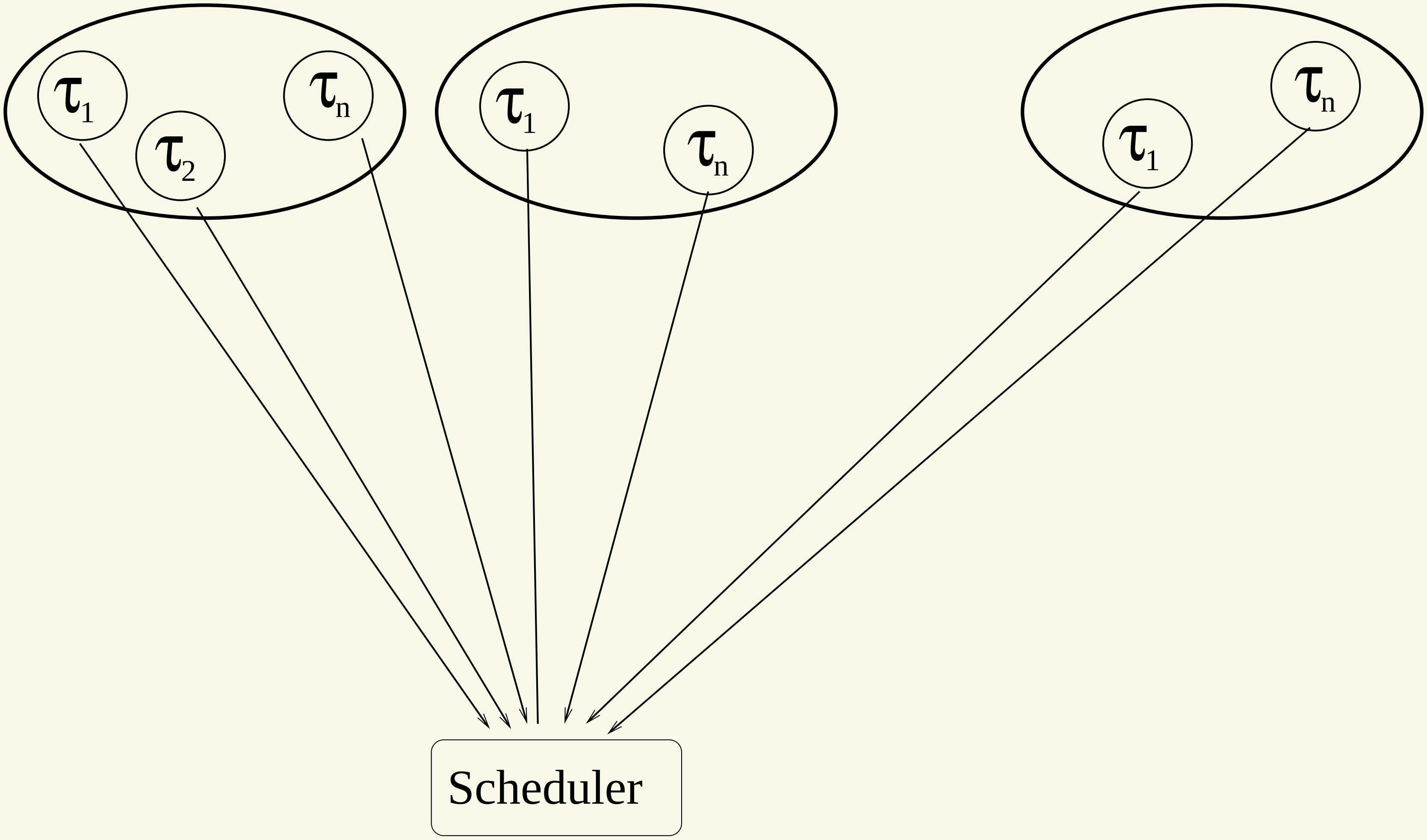
Why it is “not so smart”

- One single scheduler, that must “see” all the tasks of all the VMs
    - Internals of the VMs have to be exposed!
    - VMs cannot run their own “local” schedulers
    - Misbehaving tasks in a VM can affect other VMs
        - No isolation!!!
- Using fixed priorities might be “not so simple”
    - Think about RM: priorities in a VM might depend on other VMs...

Practical Issues

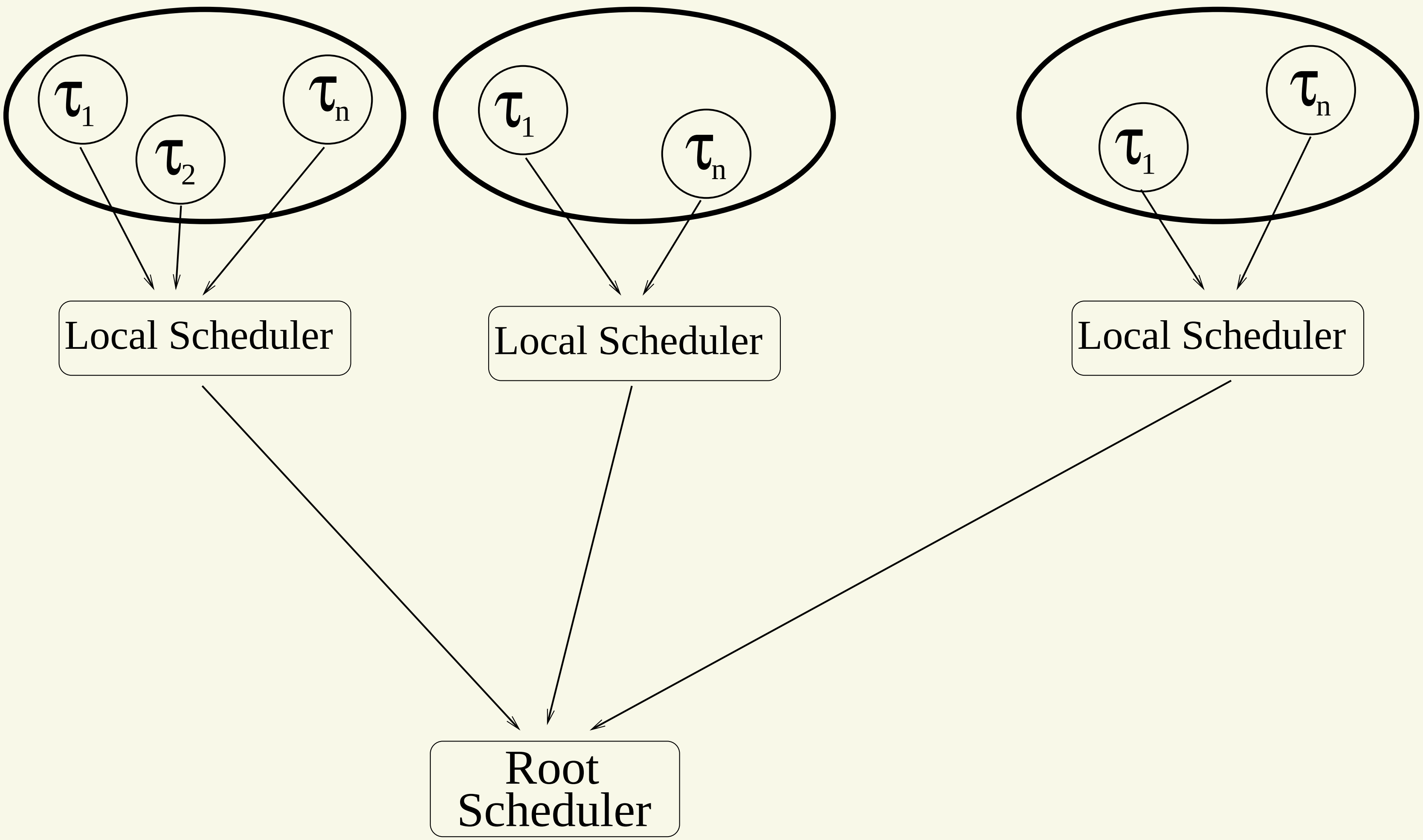
- The host/hypervisor scheduler only sees a VMs, but cannot see the tasks inside it
- Para-virtualization (of the OS scheduler) could be used to address this issue, but it is not so simple...
- ...And requires huge modifications to host, guest, and applications!
- So, how to schedule VMs?
- Two-level hierarchical scheduling system
  - Host (global / root) scheduler, scheduling VMs
  - Each VM contains its (local / 2nd level) scheduler

From a 1-Level Scheduler...



- Scheduler assigns CPU to tasks “inside the VMs”

...To a 2-Levels Hierarchy



- Host Scheduler assigns CPU to VMs
- Local Schedulers assign CPU to single tasks

Hierarchical Scheduling

- The root scheduler does not see the tasks
- The OSs inside VMs are free to define their own (fixed priorities, EDF, whatever) schedulers
  - No problems in assigning fixed priorities to tasks!
- Root scheduler: host / hypervisor scheduler
- Local scheduler: guest scheduler
- Problem: what to use as a root scheduler?
  - We must have a model for it
  - Must allow to compose the “local guarantees”
- Before going on, summary of RT definitions and concepts

Real-Time Guarantees in a Component

- First requirement: analyse the schedulability of a component independently from other components
  - This means that the root scheduler must provide some kind of temporal protection between components
- Various possibilities
  - Resource Reservations / server-based approach
  - Static time partitioning
  - ...
- In any case, the root scheduler must guarantee that each VM receives a minimum amount of resources in a time interval

Schedulability Analysis: the Basic Idea

- (Over?)Simplifying things a little bit...
- ...Suppose to know the amount of time needed by a component to respect its temporal constraints and the amount of time provided by the root scheduler
- A component is "schedulable" if

$$\text{demanded time} \leq \text{supplied time}$$

  - "demanded time": amount of time (in a time interval) needed by a component
  - "supplied time": amount of time (in a time interval) given by the root scheduler to a component
- Of course the devil is in the details

Demanded Time

- Amount of time needed by a component to respect its temporal constraints
  - Depends on the time interval we are considering
  - Depends on the component's local scheduler
    - EDF $\rightarrow dbf(t) = \sum_j \max\{0, \left\lfloor \frac{t+T_j-D_j}{T_j} \right\rfloor\} C_j$
    - RM: $\rightarrow$ workload $W(t) = C_i + \sum_{j<i} \left\lceil \frac{t}{T_i} \right\rceil C_j$
  - Note: $W(t)$ is very pessimistic, $dbf(t)$ is not
- This is the description of the temporal requirements of a component we were searching for...
- And what about the supplied time?

Supplied Time

- Description of the root scheduler temporal behaviour
- More formally:
  - Depends on the time interval t we are considering
  - Depends on the root scheduler $\mathcal{A}$
- Minimum amount of time given by $\mathcal{A}$ to a VM in a time interval of size s
  - Given all the time interval $(t_0, t_1) : t_1 - t_0 = s$...
  - ...Compute the size of the sub-interval in which $\sigma(t) = VM$...
  - ...And then find the minimum!

Supplied Time Bound Function

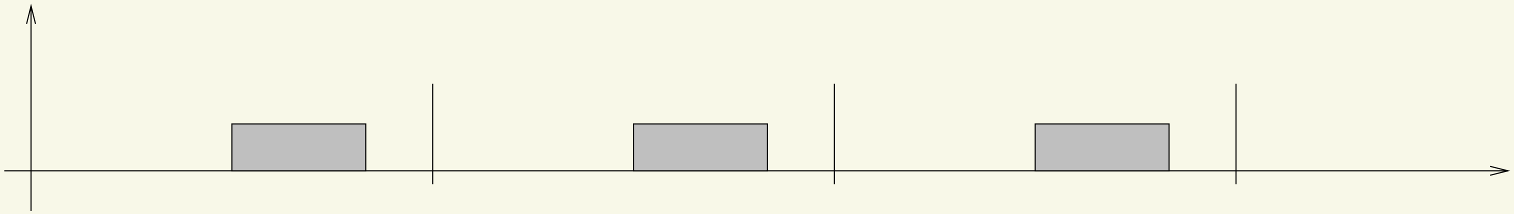
- Even more formally:
  - Define $s(t) = \begin{cases} 1 & \text{if } \alpha(t) = VM \\ 0 & \text{otherwise} \end{cases}$
  - Time for VM in $(t_0, t_0 + s)$: $\int_{t_0}^{t_0+s} s(t)dt$
  - Then, compute the minimum over t_0
- $sbf(t) = \min_{t_0} \int_{t_0}^{t_0+t} s(x)dx$

Example: Static Time Partitioning

- First (very simple) example of VM scheduling: static time partitioning
  - Static schedule describing when time is assigned to each VM
  - Pre-computed $\sigma(t)$
- Generally, periodic!
  - Otherwise, need to store an infinite schedule...
  - ...Might be problematic!
- Example: VM$_\mathcal{A}$ is scheduled in $(3, 4)$, $(9, 10)$, $(15, 16)$, ...
  - More formally: $s(t) = 1$ if $6k + 3 \le t \le 6k + 4$, $s(t) = 0$ otherwise

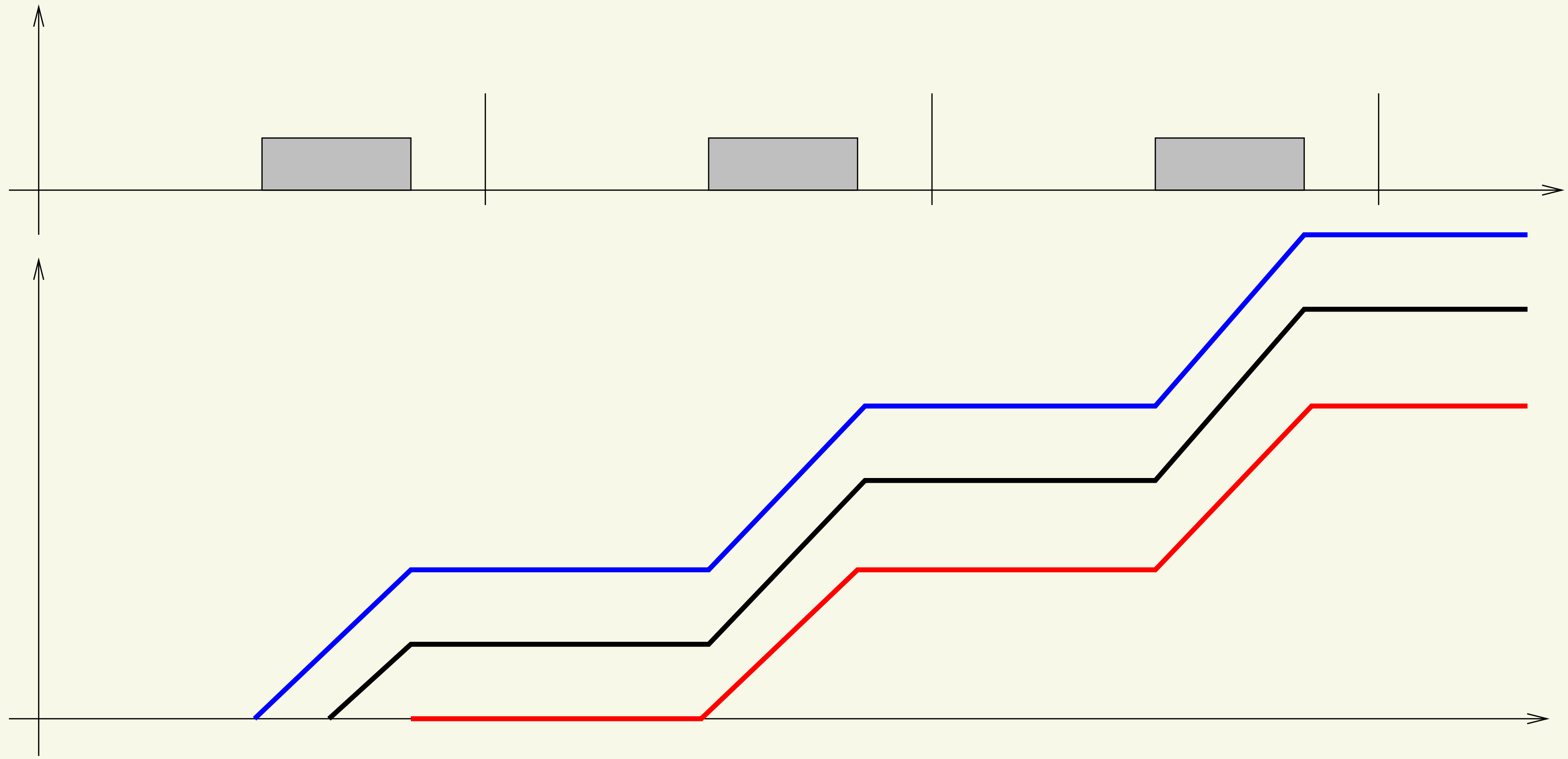
Example: Static Time Partitioning - 2

$$s(t) = \begin{cases} 1 & \text{if } 6k+3 \leq t \leq 6k+4 \\ 0 & \text{otherwise} \end{cases}$$



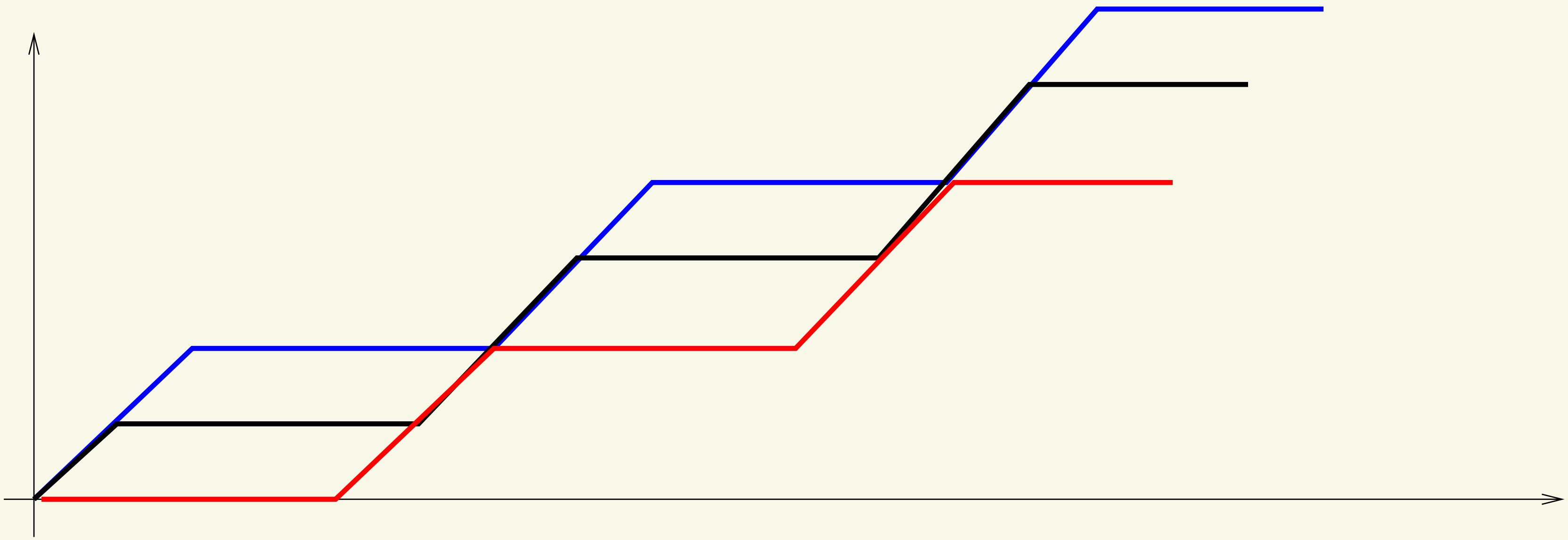
- What is the supply bound function $sbf(t)$ in this case?
- Let's try different supply functions compatibe with this schedule...
- ...And see what is the worst case!
  - Intervals of size t starting at different times...

Example: Static Time Partitioning - 3



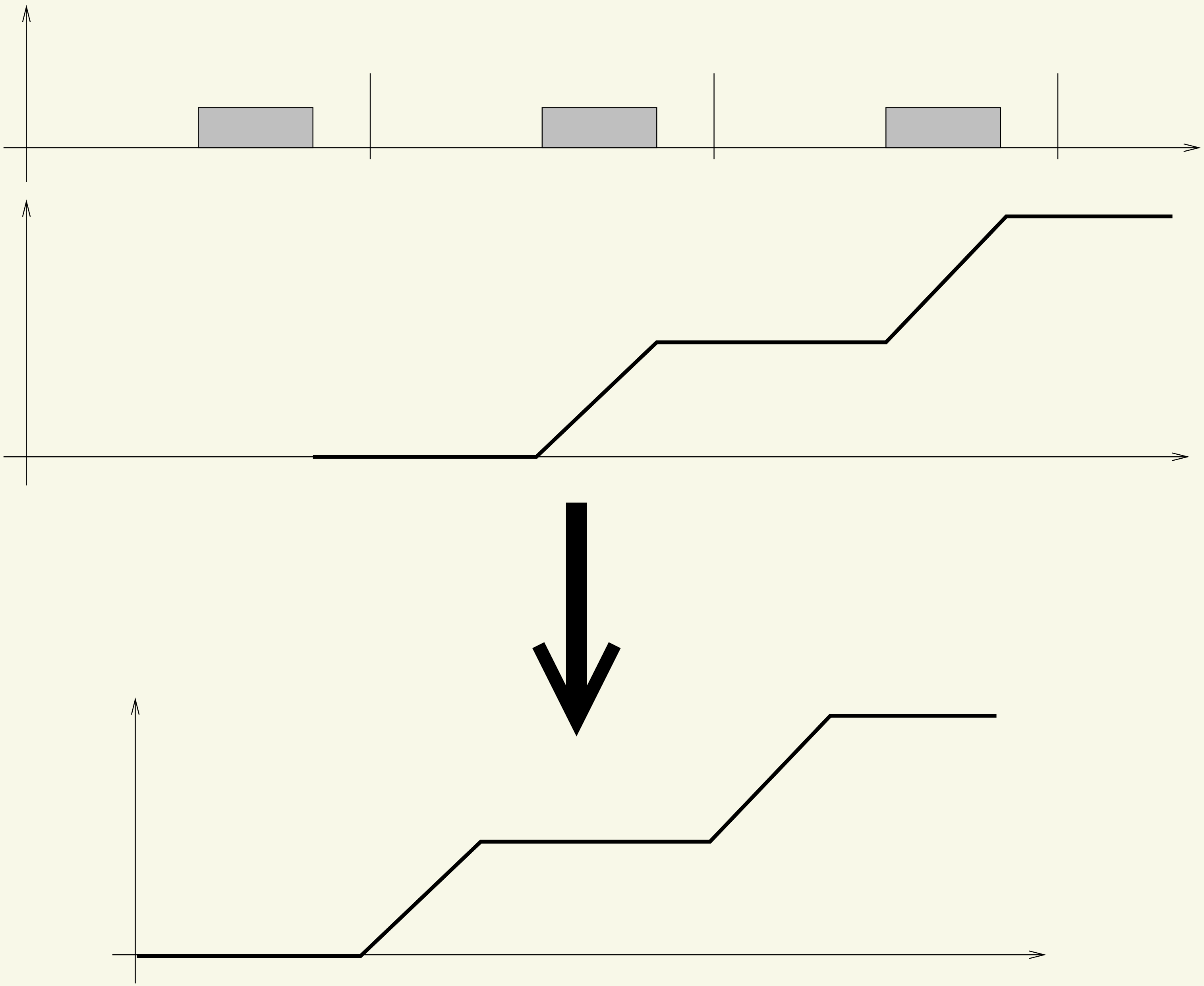
- Different supply functions depending on when the considered interval begins
- Which one is the worst case (supply bound function)?

Example: Static Time Partitioning - 4



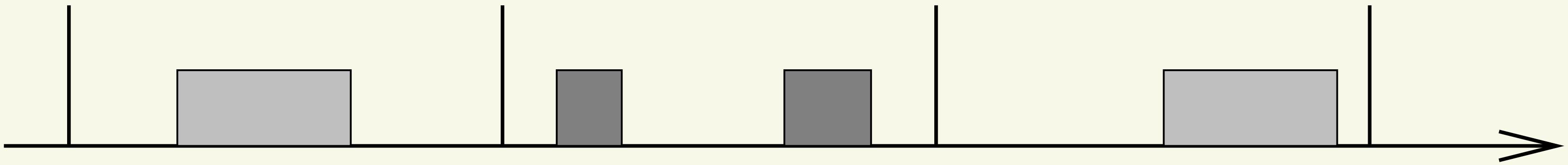
- Different supply functions depending on when the considered interval begins
- Which one is the worst case (supply bound function)?
  - The red one!

Example: Static Time Partitioning - 5



Periodic Servers

- Periodic Server $\mathcal{S} = (Q, P)$: guarantees Q units of time every period P
  - Can be implemented in different ways (example: CBS)
- Different from static allocation: we do not know where in the period the Q time units are allocated
  - Execution inside a period can even be preempted!

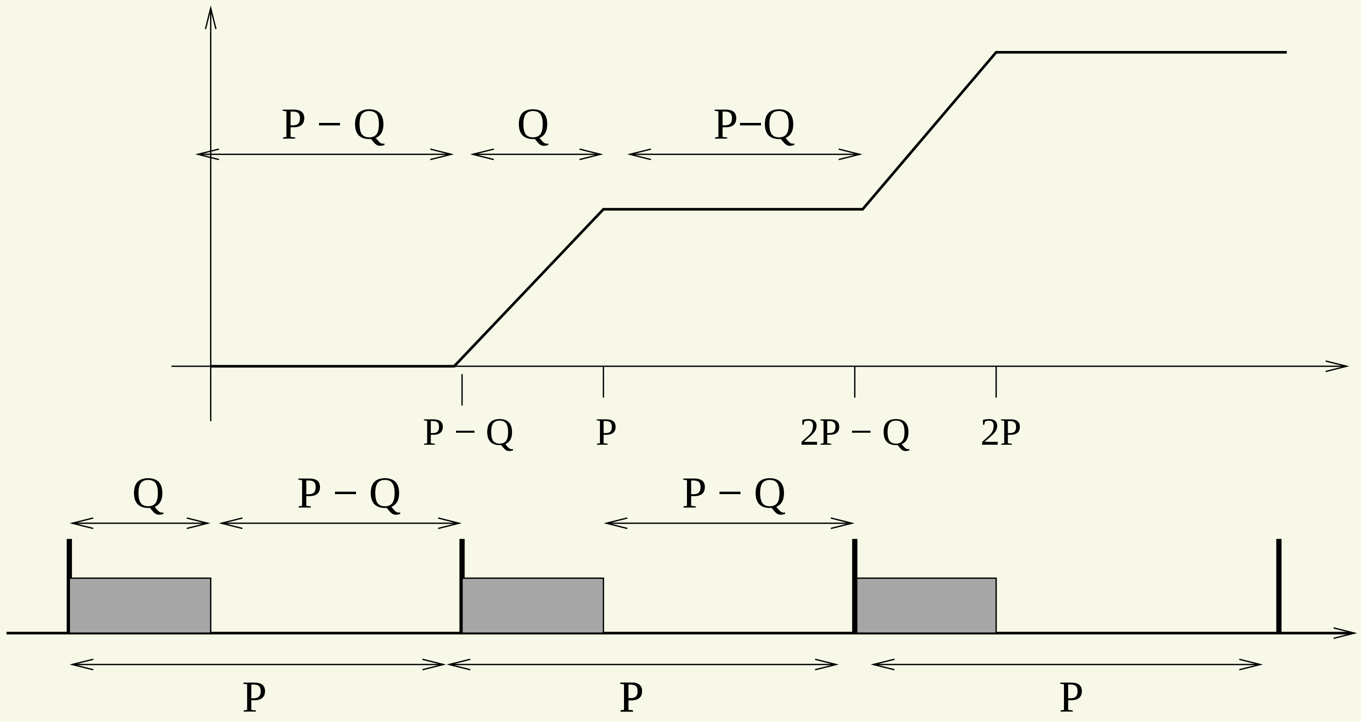


Periodic Servers — Supplied Time

- $sbf(t)$: minimum amount of time that a VM is guaranteed to receive in a time interval of size t
  - Consider all the possible intervals of size t...
    - As already seen for static time partitioning
  - ...And all the possible “legal CPU allocations” generated by the periodic server!
- Big difference with static time partitioning: consider all the possible allocations of Q in the period

The Wrong Solution

- Immagine Q is allocated at the beginning of the period
  - Worst case allocation: $t0$ immediately after Q
  - The time interval starts when the root scheduler deschedules the component



The Wrong Solution — 2

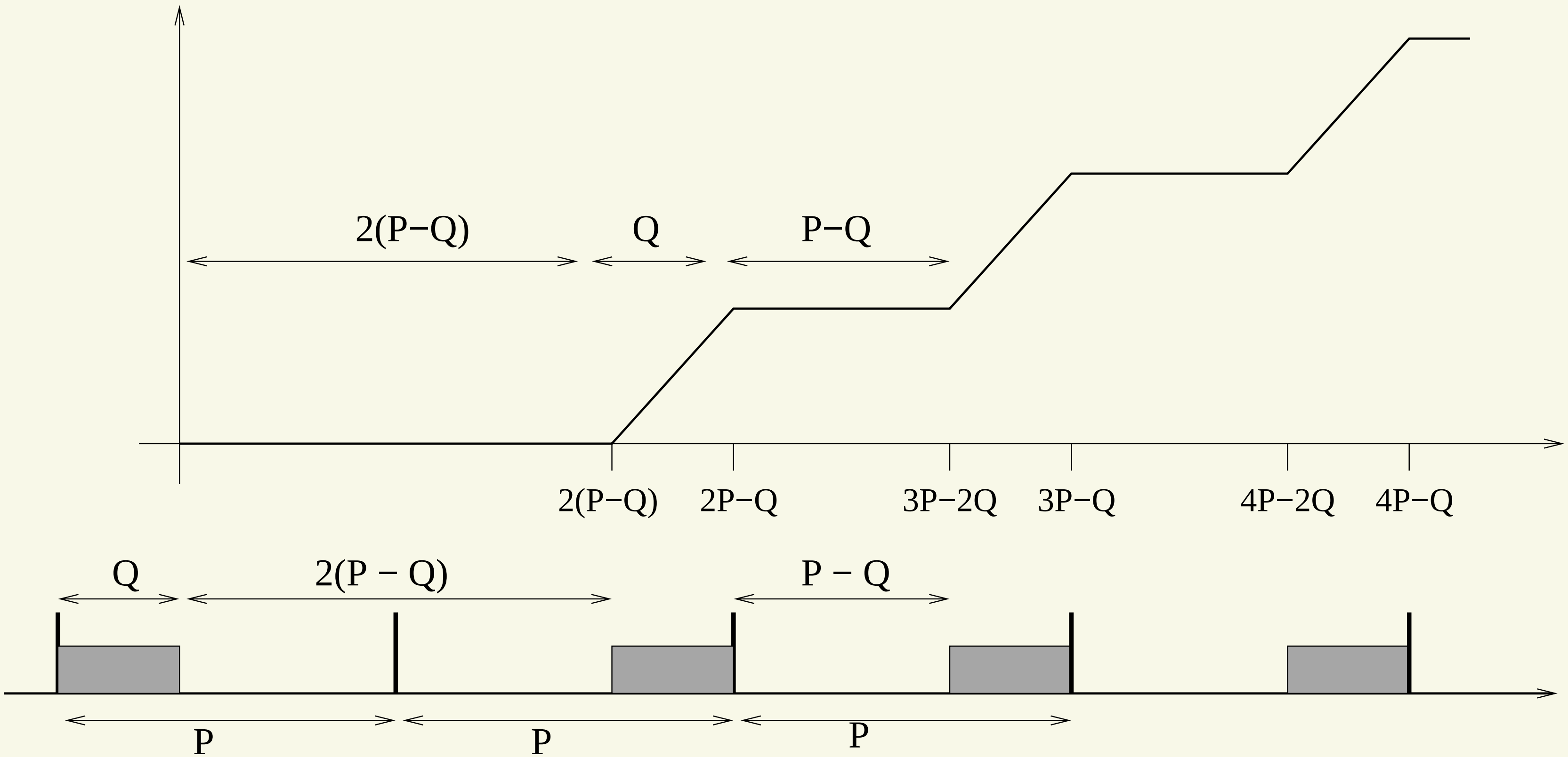
- Supplied time: 0 until $P - Q$...
- ...Then increases with slope 1 until P...
- ...Then flat again until $2P - Q$...
- ...

$$sbf(t) = \begin{cases} 0 & \text{if } t < (P - Q) \\ (n-1)Q & \text{if } (n-1)P \le t < nP - Q \\ t + nQ - (n-1)P & \text{if } nP - Q \le t < nP \end{cases}$$

Why Wrong?

- The previous computation assumed Q always at the beginning of a period...
- ...But this is not the worst case!
  - Think about the second period...
  - ...What happens if the root scheduler delays the allocation?
  - The initial "0 allocation period" increases!!!
- Worst-case schedule: Q at the beginning of the first period and at the end of the second one
  - See the difference with static time partitioning?

Considering the Worst-Case Situation



$$sbf(t) = \begin{cases} 0 & \text{if } t < 2(P-Q) \\ (n-1)Q & \text{if } nP - Q \le t < (n+1)P - 2Q \\ t - (n+1)(P-Q) & \text{if } (n+1)P - 2Q \le t < (n+1)P - Q \end{cases}$$

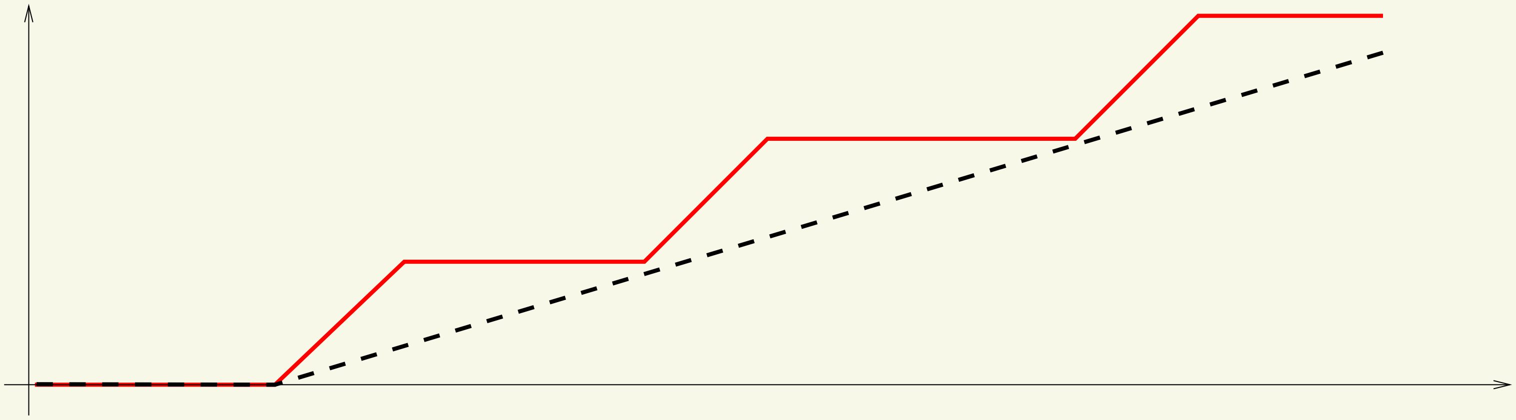
Understanding the Supplied Bound Function

- Supplied bound function $sbf(t)$: minimum amount of time that a VM is guaranteed to receive in a time interval of size t
  - Considers all the possible intervals of size t...
- Strange looking function!
  - Flat for large intervals of time...
  - $\frac{\delta sbf(t)}{\delta t} = 1$ in the other intervals
- Can we “summarise” it with something simpler?
- What about a line ($y = ax + b$)?
  - $sbf(t) < 0$ makes no sense...
  - So, better $sbf(t) = max\{0, at + b\}$

A Linear Approximation

- $sbf(t) = max\{0, at + b\}$... $at + b$ is below 0 for $t < -b/a$
  - Let's rewrite the equation... $at + b = a(t - \Delta)$ with $\Delta = -b/a$

$$sbf(t) = \begin{cases} 0 & \text{if } t < \Delta \\ a(t - \Delta) & \text{otherwise} \end{cases}$$



Interpreting the Linear Approximation

- $t < \Delta \Rightarrow sbf(t) = 0$: Δ is the *allocation delay* for the VM
  - Worst-case delay between the VM becoming active and the root scheduler scheduling it
  - How much time should I wait before the root scheduler starts giving the CPU to my VM?
- a (sometimes referred as α) is the *bandwidth* of the VM
  - Minimum fraction of CPU time reserved for the VM after the initial delay
- Of course, (a, Δ) should be so that $a(t - \Delta)$ is below the real $sbf()$

Periodic Servers Revisited

- How to compute (a, Δ) for a periodic server (Q^s, T^s)?
  - $a = \frac{Q^s}{T^s}$, $\Delta = 2(T^s - Q^s)$
- So, after the initial delay $2(T^s - Q^s)$ the VM is really receiving the expected fraction of CPU time (Q^s/T^s)
  - If we reduce T^s (keeping Q^s/T^s unchanged)...
  - ...$sbf(t)$ tends to the "fluid allocation"!
- Why not using very very small server periods?
  - Of course there is a reason...

The Design Problem

- Given a component (set of tasks and a local scheduler)...
  - Described by a time demand function (workload for fixed priorities)
- ...Find a root scheduler (and scheduling parameters) able to respect the components' temporal constraints
- Problem reduced to solving "$sbf(t) \geq dbf(t)$" for a set of points
  - Must be verified for all the points in case of EDF
  - Must be verified for at least one point in case of fixed priorities

Simplified Design

- $sbf(t) \geq dbf(t)$
- Using $sbf(t) = a(t - \Delta)$...

$$a(t - \Delta) \geq dbf(t) \Rightarrow \Delta \leq t - \frac{dbf(t)}{a}$$

- Solve this for every $(t, dbf(t))$, and plot the solution on a $a - \Delta$ plane...
- ...Then compute the intersection (for EDF) or union (for fixed priorities)