Arm Timers and Fire!

KVM Forum 2018

Christoffer Dall
christoffer.dall@arm.com

Introduction

KVM/Arm currently provides basic timekeeping functionality:

- VMs can read a counter to measure passing of time
- VMs can program and cancel timers

Without trapping to the hypervisor. Yay!

What are we missing

No accounting for pausing the VM or suspending the system:

- Results in warnings from the guest OS

No accounting for stolen time:

- Guest processes are starved
- Warnings from the guest OS when oversubscribing physical CPUs

Migrating to a new physical machine with a different counter frequency:

- Timekeeping not aligned with software expectations

Background – Arm Generic Timers Architecture

Also known as the “Arch Timers” or “Architected Timers”

In Armv8.0, we have:

- The physical counter
- The virtual counter
- Four timers:
  - EL3 Physical Timer (for Secure World – not relevant for KVM)
  - EL2 Physical Timer (for the hypervisor)
  - EL1 Physical Timer (for the OS)
  - EL1 Virtual Timer (for the OS)

Background – Arm Generic Timers Architecture

In **Armv8.1**, with the Virtualization Host Extensions (VHE), we have:

- The physical counter
- The virtual counter
- Five timers:
  - EL3 Physical Timer (for Secure World – not relevant for KVM)
  - EL2 Physical Timer (for the hypervisor)
  - **EL2 Virtual Timer (for the hypervisor)**
  - EL1 Physical Timer (for the OS)
  - EL1 Virtual Timer (for the OS)

arm

Background – Arm Generic Timers Architecture

What is a Counter and a Timer?

Counter

Simple 64-bit value monotonically increasing at a per-system specific frequency.

Timer

A device that triggers an event after some time.

Each timer has:

- CVAL (Compare Value)
- CTL (Control Register:
  enable, mask, status)

A timer is associated with a counter.

Asserts output line when:

```
Counter >= CVAL
```


arm

The Counters

Physical Counter is 64-bit monotonic counter

Accessed via CNTPCT_EL0

Trappable to EL1 and EL2

Virtual Counter = Physical Counter - Offset

Offset is controlled by EL2 (Hypervisor) using CNTVOFF_EL2

Accessed via CNTVCT_EL0

- Optionally trapped by OS
- VHE hypervisors read CNTVCT_EL0 with a fixed offset of zero, but the virtual timer still uses CNTVOFF_EL2 offset

Background – Arm Generic Timer Architecture

More information: Arm Architecture Reference Manual (Arm ARM)

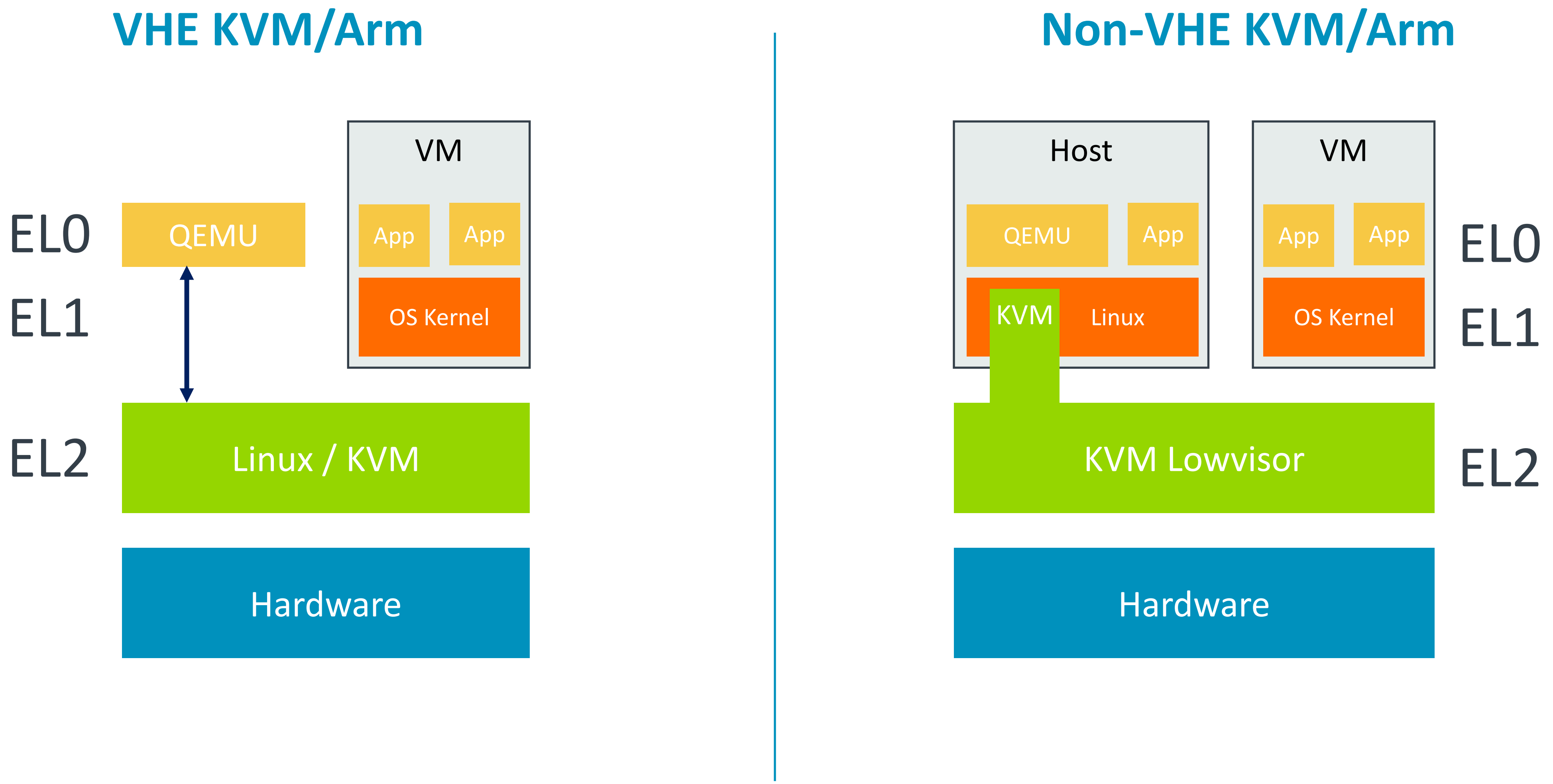
Timer	Counter	Access	Trappable by OS ?	Trappable by Hypervisor?
EL2 Physical Timer	Physical Counter	EL2	-	-
EL2 Virtual Timer	Physical Counter	EL2	-	-
EL1 Physical Timer	Physical Counter	EL2, EL1, EL0	yes	yes
EL1 Virtual Timer	Virtual Counter	EL2, EL1, EL0	yes	no

Myth: "The virtual timer directly generates virtual interrupts" – not true!

Arm Generic Timers and KVM/Arm Today

arm

(Very Quick) Refresher on KVM/Arm

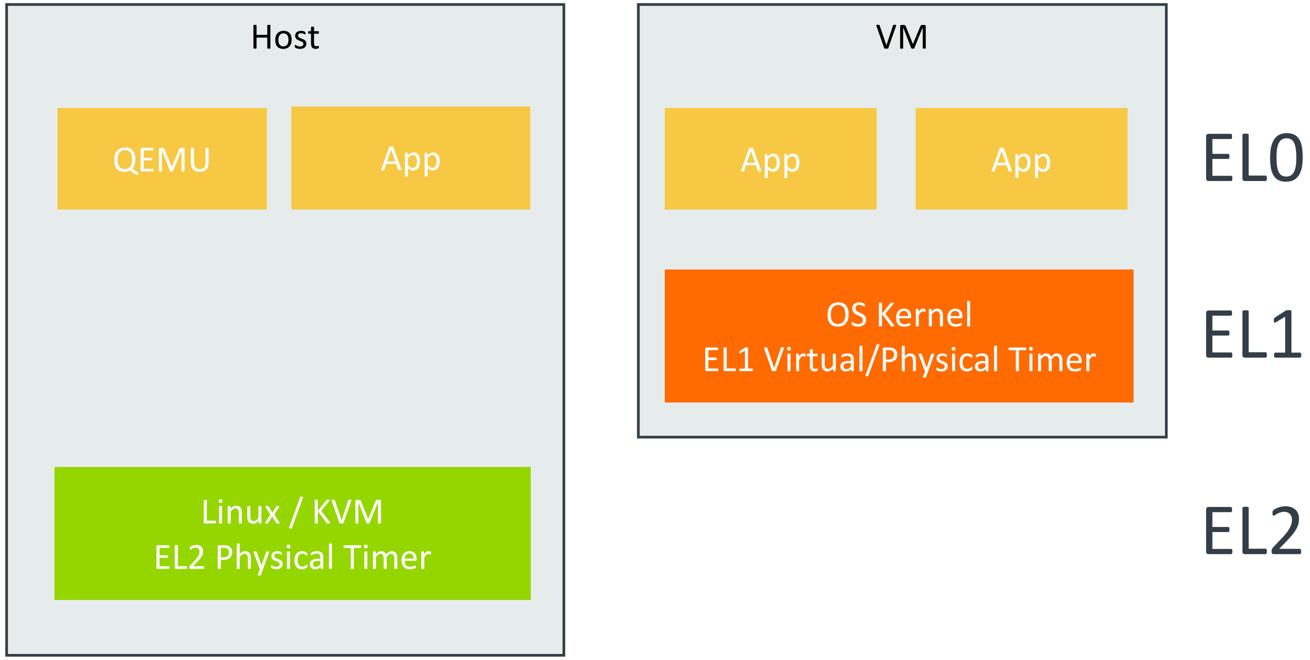


KVM/Arm and Generic Timer: VHE

KVM and Linux Host use
  EL2 Physical Timer

Guest uses:
  EL1 Virtual Timer and/or
  EL1 Physical Timer

EL2 Virtual Timer is not used



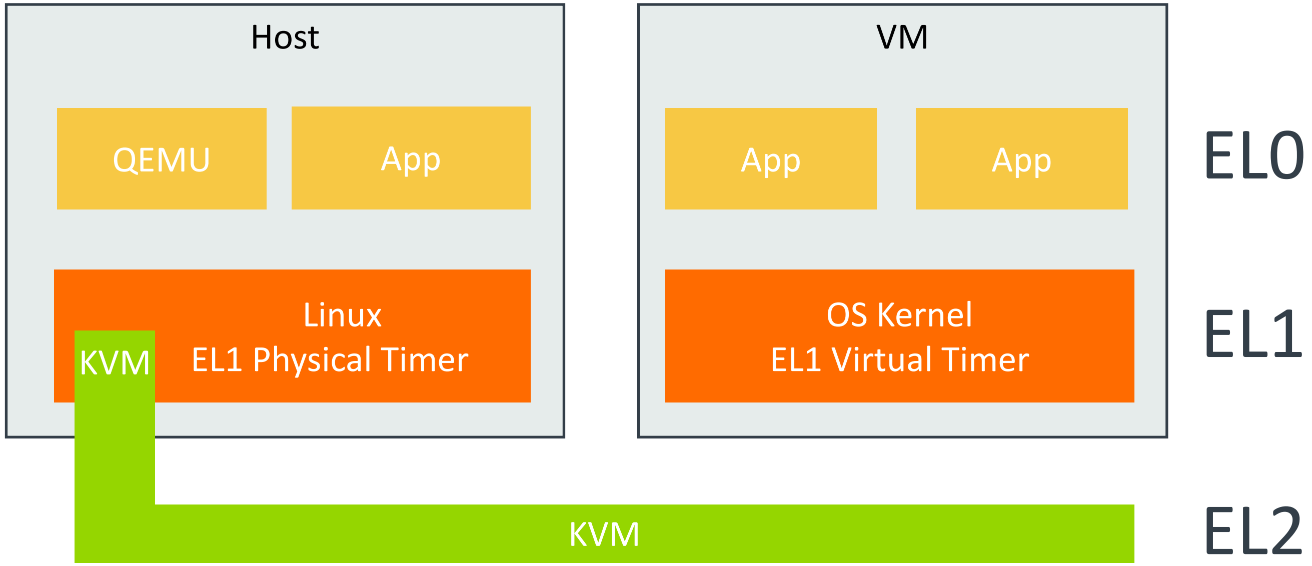
KVM/Arm and Generic Timer: Non-VHE

Host uses
- EL1 Physical Timer

Guest uses
- EL1 Virtual Timer
- EL1 Physical Timer (trap-and-emulate)

EL2 Physical Timer is not used

EL2 Virtual Timer may not exist



arm

Linux and Generic Timer Access

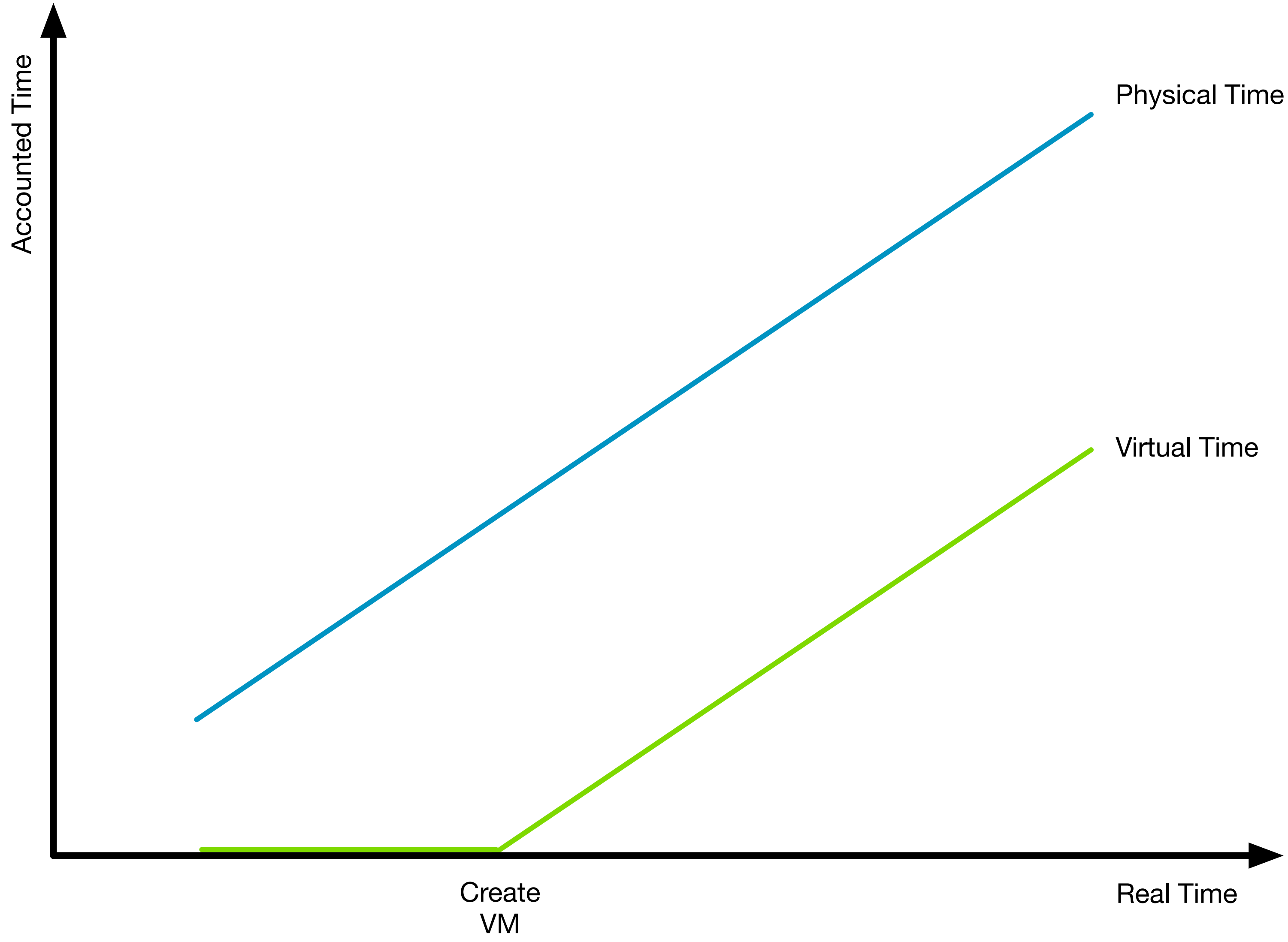
Timer	VHE KVM/Arm	Non-VHE KVM/Arm
EL1 Physical Timer	Direct Access	Trap-and-emulate
EL1 Virtual Timer	Direct Access	Direct Access

What does this mean for Linux?

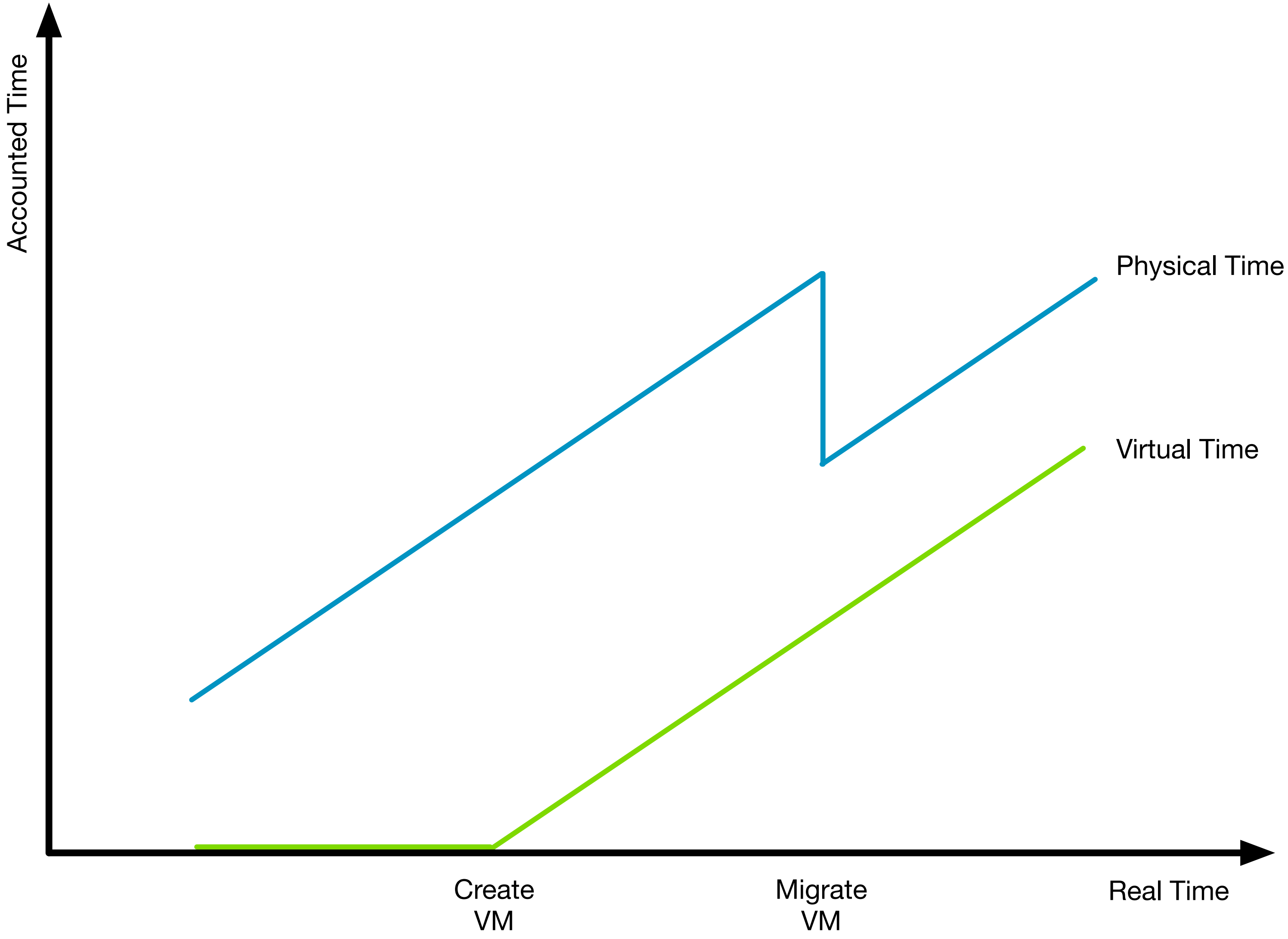
Linux can observe **some** view of both virtual and physical time, but Linux today always uses the virtual counter/timer when running as a guest.

So we have an offset to play with.

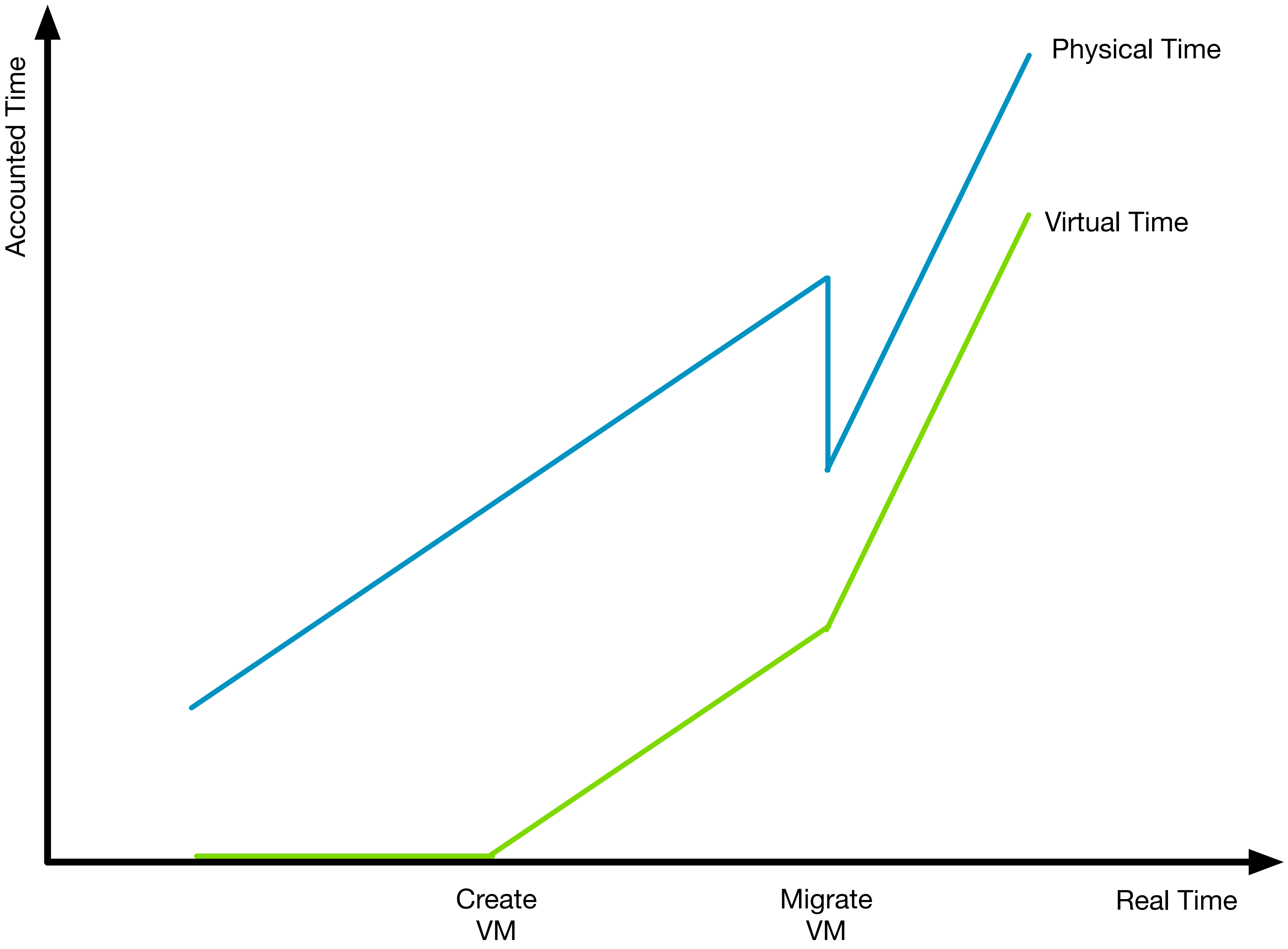
KVM/Arm and the Virtual Counter



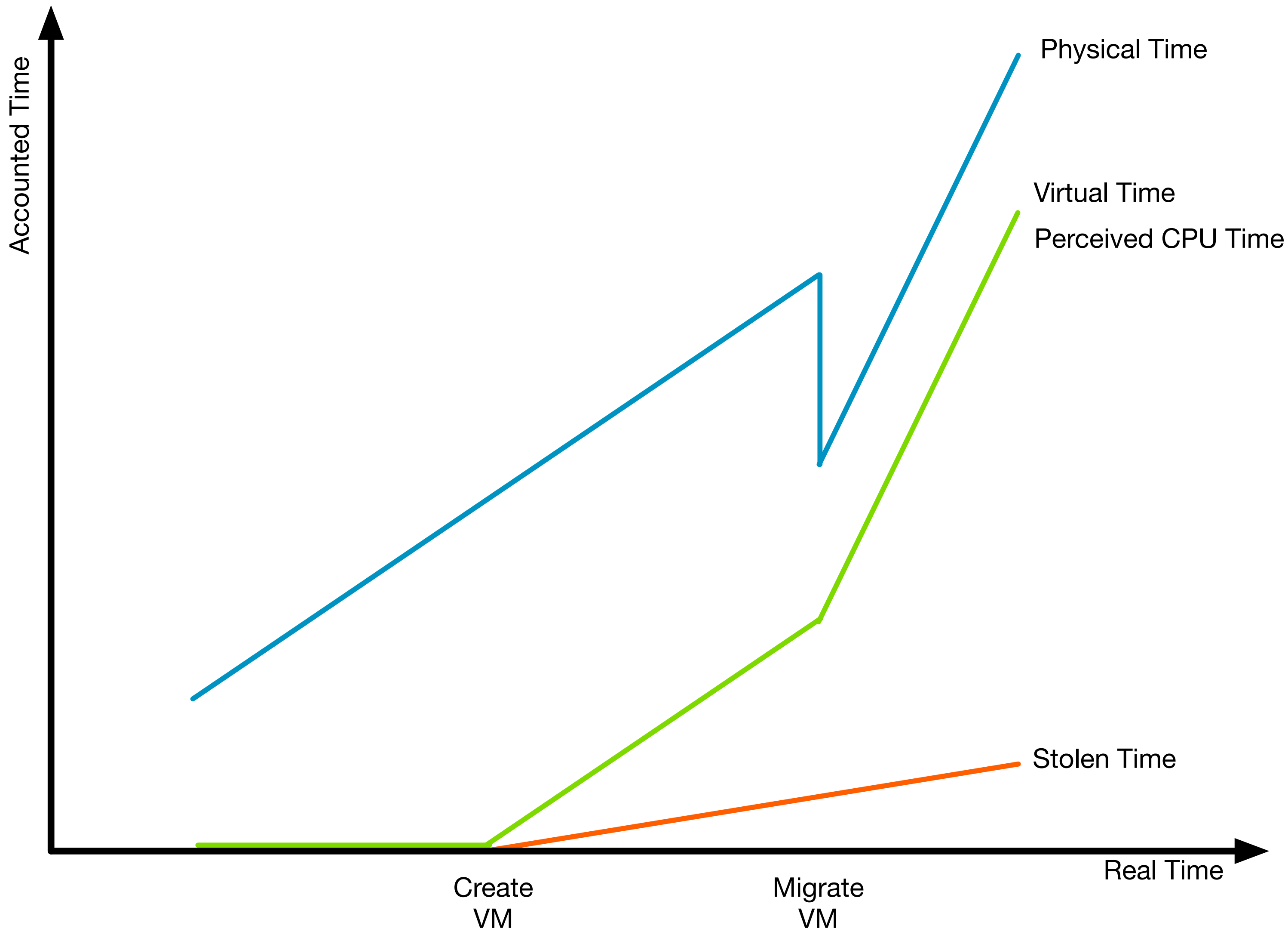
KVM/Arm and the Virtual Counter – Migration



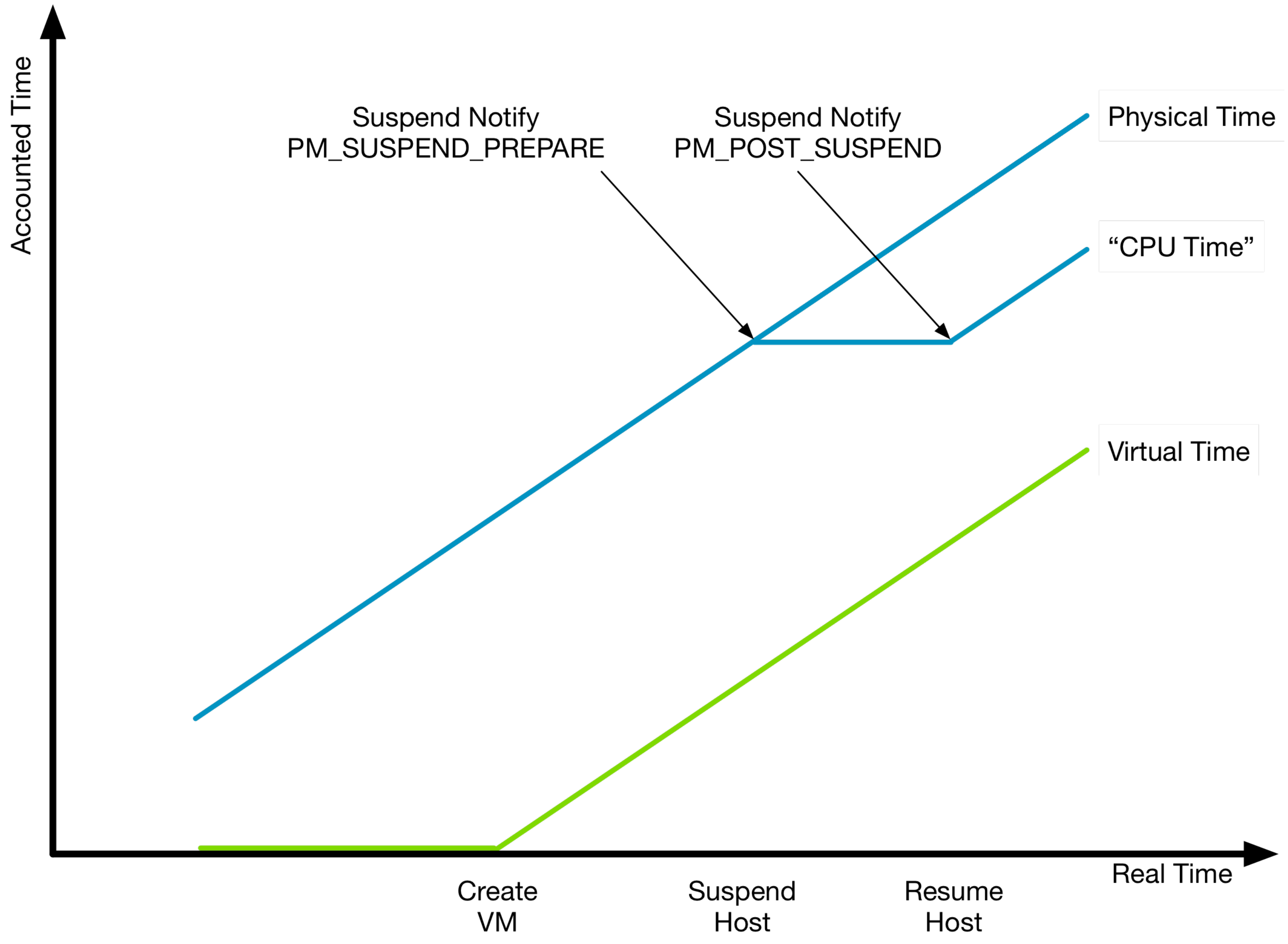
KVM/Arm and the Virtual Counter – Bad Migration



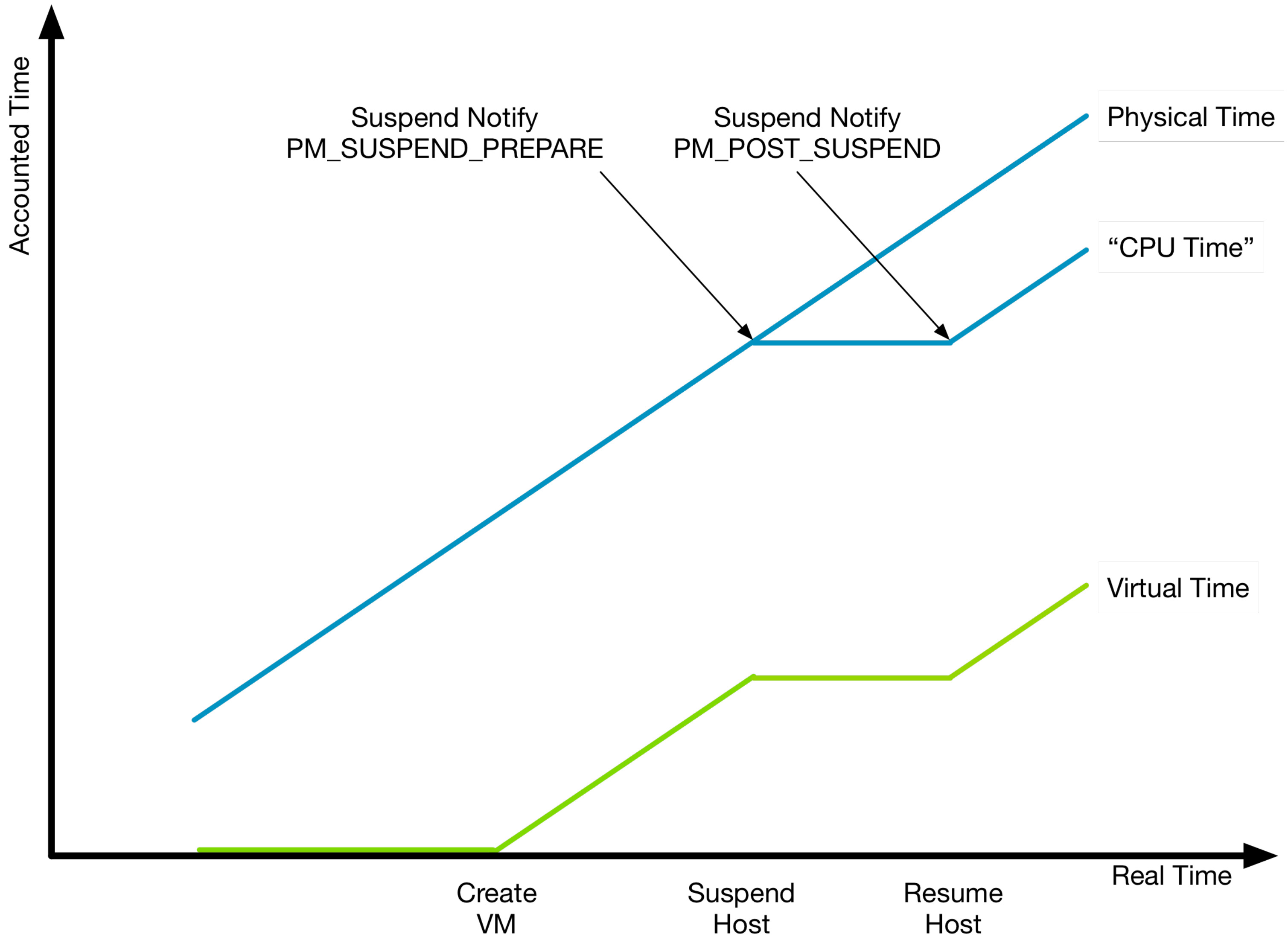
KVM/Arm and the Virtual Counter – Stolen Time



KVM/Arm and the Virtual Counter – Suspend



KVM/Arm and the Virtual Counter – Pause while suspended



What have we learnt so far

Basic timekeeping for VMs is ok

Migration:

- OK with the same counter frequency
- Weird with different counter frequencies

We are not accounting for stolen time at all

No suspend notifications in VMs

- But we can fix this in KVM today

Paravirtualized Time

Paravirtualized Time for Arm-based Systems

https://developer.arm.com/docs/den0057/a

BETA software interface specification from Arm

Provides unified interface between hypervisor and guest OS for PV time

Feature is discoverable via SMCCC v1.1

Standardizes hypercall numbers, parameters, return codes, and data structures

arm

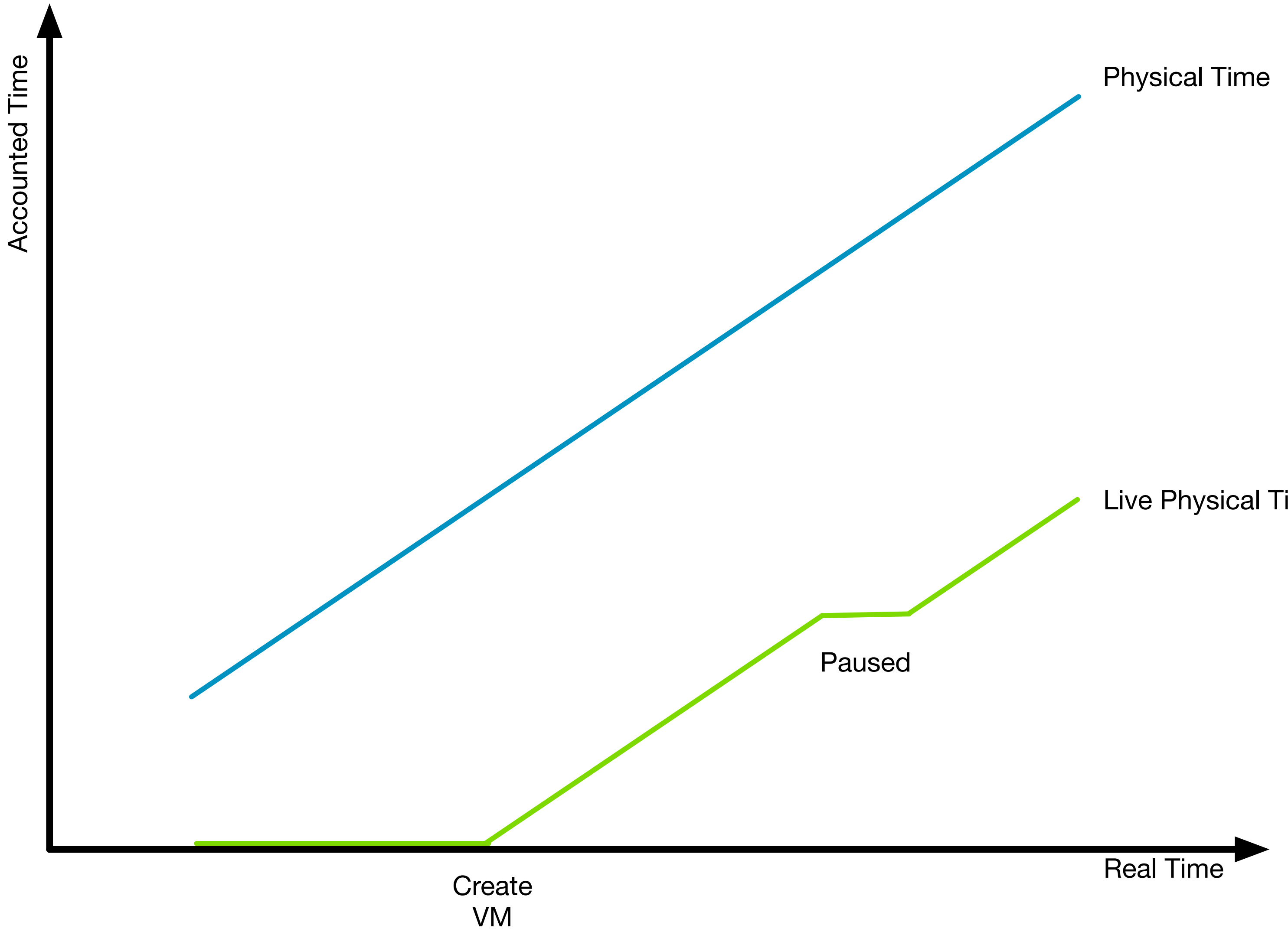
Definitions of Time

Paused:	VM is deliberately paused or host is not running
Running:	VM not paused and VCPUs *could be* scheduled.

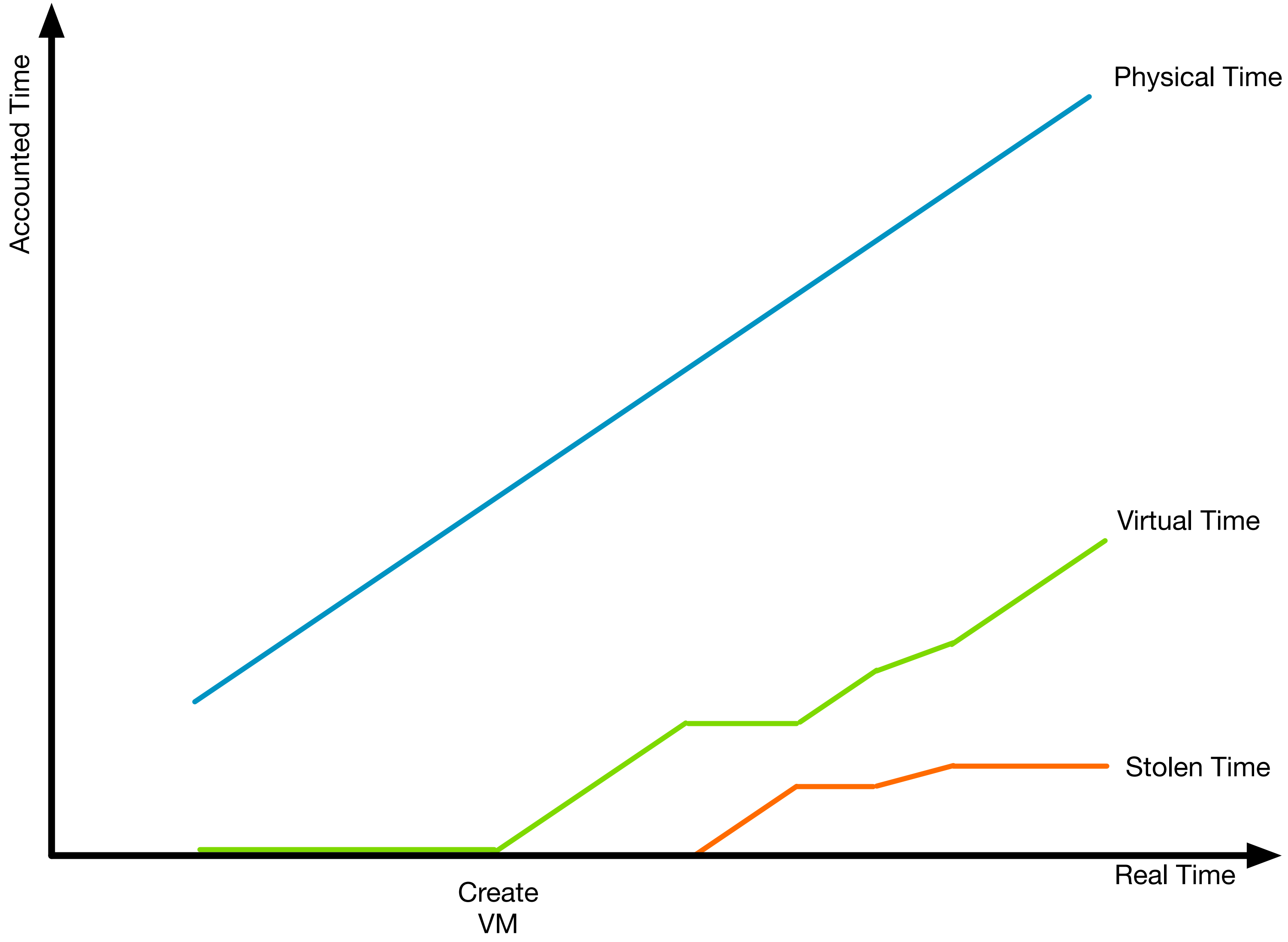
Physical Time:	Real time
Live Physical Time (LPT):	Physical Time - Paused
Virtual Time:	VCPU running (or deliberately waiting for interrupts)
Stolen Time:	VCPU runnable, but queued waiting for other tasks

Fpv:	Paravirtualized frequency

Live Physical Time

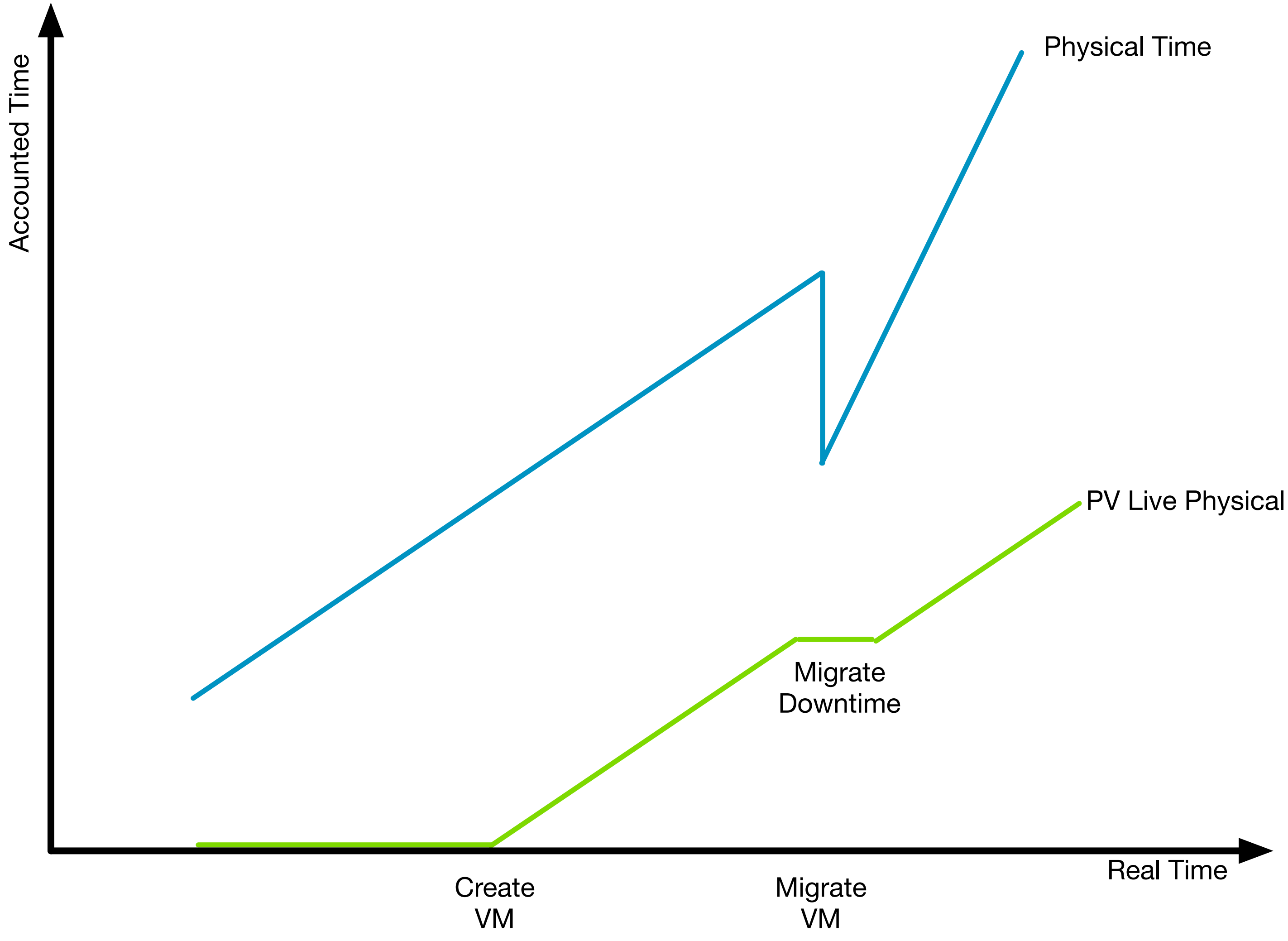


Virtual and Stolen Time



arm

Migrating with different frequencies



Frequencies

Problem reminder: Migrating VMs across machines with different counter frequencies

Fpv: PV Frequency chosen by the hypervisor

Fn: Native frequency of a system

$$\text{Time PV} = \text{Counter} * (\text{Fpv} / \text{Fn})$$

arm

How does the VM know Fpv and Fn?

Shared data structure between host and guest:

```
struct pv_time_lpt {
	...
	u64 sequence_number;            // consistency
	u64 scale_mult;                 // Fn -> Fpv conversion
	u32 shift;                      // Fn -> Fpv conversion
	u64 Fn;                         // Frequency native
	u64 Fpv;                        // Frequency PV
	u64 div_by_fpv_mult;            // Fpv -> Fn conversion
	...
};
```

How does this work?

```
extern struct pv_time_lpt *ptv;

u64 live_physical_time()
{
      u64 x;
      u32 s_before, s_after;
      do {
            s_before = ptv->sequence_number;
            x = scale_to_fpv(CNTVCT_EL0); // read virtual counter
            s_after = ptv->sequence_number;
      } while (s_after != s_before);
      return x;
}
```

How does this work?

What we are trying to do:

PV Time = vcount * (Fpv / Fn)

```
extern struct pv_time_lpt *ptv;

u64 scale_to_fpv(u64 vcount)
{
	/* In AArch64 this can be achieved with a shift and a
	 * UMULH instruction. */

	u128 tmp = ptv->scale_multiplier * (vcount << ptv->shift);

	return tmp >> 64;
}
```

Fast timekeeping in Fpv without trapping!

Programming Timers in a PV World

We now keep PV time, ticking at Fpv

But we have to program a hardware timer, which uses Fn

$$\text{Interval PV} = \text{Interval} * (\text{Fn} / \text{Fpv})$$

To avoid rounding down:

$$\text{Interval PV} = (\text{Fn} * \text{Interval} + \text{Fpv} - 1) / \text{Fpv}$$

```
int upscale_to_native(u64 interval)
{
	u64  x = ptv->Fn * interval + ptv->Fpv - 1;
	u128 y = ptv->div_by_fpv_mult * x;
	return y >> 64;
}
```

PV Stolen Time

Shared per-VCPU data structure between host and guest:

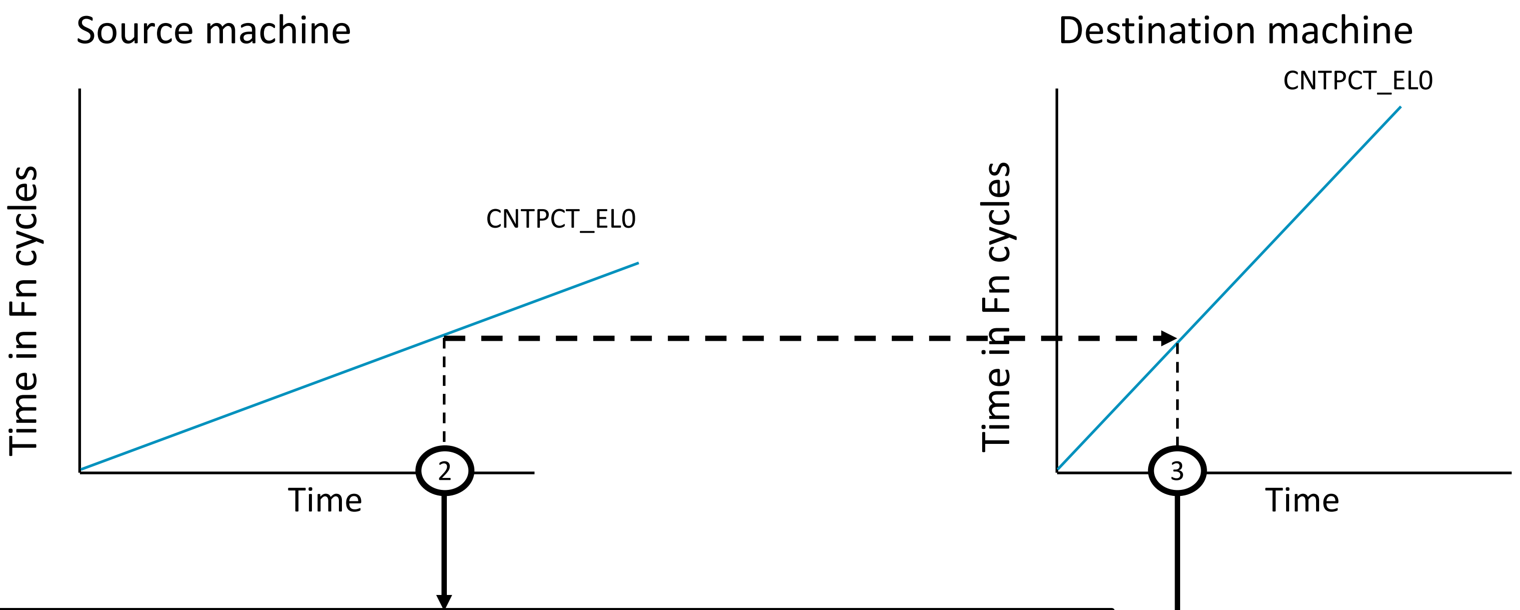
```
struct pv_time_vcpu_stolen {
      ...
      u64 stolen_time;          // stolen time in ns
      ...
};
```

No sequence_number. Stolen time accessed using 64-bit single-copy atomics.

Migration - Counter

We adjust the offset that expresses paused time when the native frequency changes.

1. Store LPT Native and Fn source
2. Scale LPT Native to Fn destination
3. Calculate new offset on destination machine.



VM departs from source machine:

```
LPT_Native = CNTPCT_EL0 – CNTVOFF_EL2_guest
Fn_source  = Fn
```

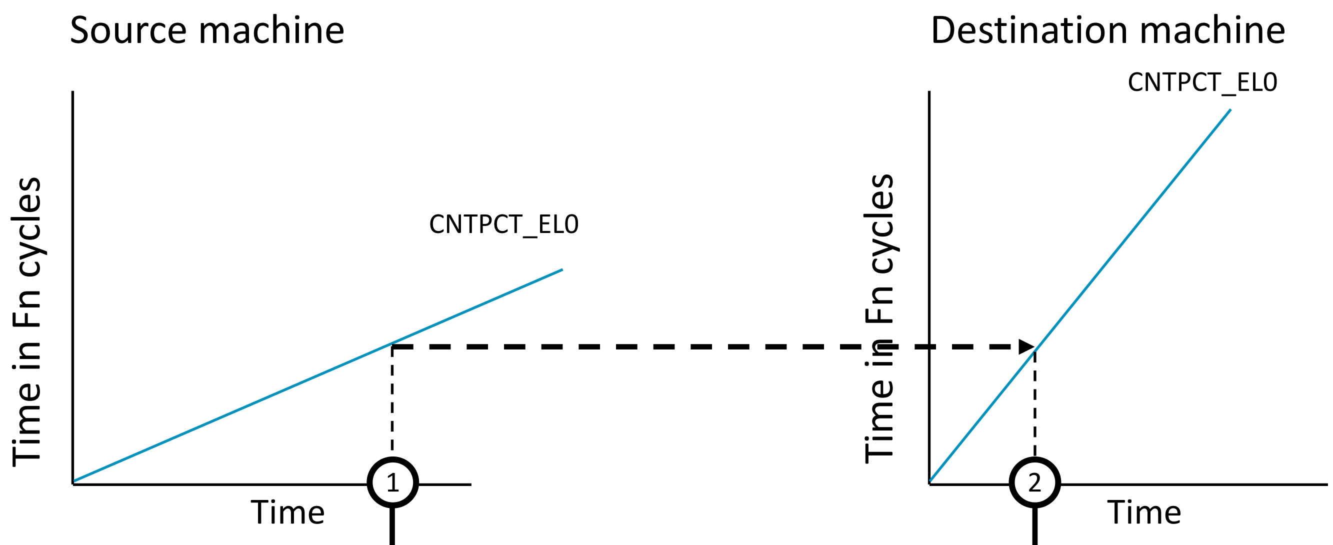
Prior to scheduling VM in destination machine:

```
LPT_Native = LPT_Native*Fn/Fn_source
shift = ceil(log2(Fpv/Fn))
scale_mult = 2^(64-shift)*Fpv/Fn
sequence_number++
CNTVOFF_EL2_guest = CNTPCT_EL0 - LPT_Native
```

Migration - Timer

We adjust the Compare Value of timers.

1. Store interval native and Fn source
2. Scale interval native to Fn destination
3. Calculate new Compare Value on destination machine.



VM departs from source machine:

```
vt_interval  = CNT_CVAL_EL0 -
               (CNTPCT_EL0 - CNTVOFF_EL2_guest);
Fn_source   = Fn
```

Prior to scheduling VM in destination machine:

```
vt_interval  = vt_interval * (Fn / Fn_source)
CNT_CVAL_EL0 = (CNTPCT_EL0 - CNTVOFF_EL2_guest) +
               vt_interval
```


Putting it all together

https://developer.arm.com/docs/den0057/a

Migration:

- Fpv to make sense of time across systems with different counter frequencies
- Efficient software adjustment with the use of scale+shift
- Timers can be programmed with reverse adjustment
- Values are adjusted by hypervisor when migrating across physical machines

Suspend:

- Host machine suspend or VM migration downtime corrections expressed via Live Physical Time (LPT)
- LPT expressed via CNTVOFF_EL2 and the virtual counter

Stolen Time:

- Hypervisor provides stolen time via shared data structure

Paravirtualized Time
… and nested virtualization

Nested Virtualization: Oh no, more turtles…

Arm Generic Timer Architecture not designed with nested virtualization in mind

Complexity space explodes.

Guest hypervisor can be VHE/non-VHE

Host hypervisor can be VHE/non-VHE

Guest hypervisor can use PV time itself, or not, can expose PV time to guests, or not…

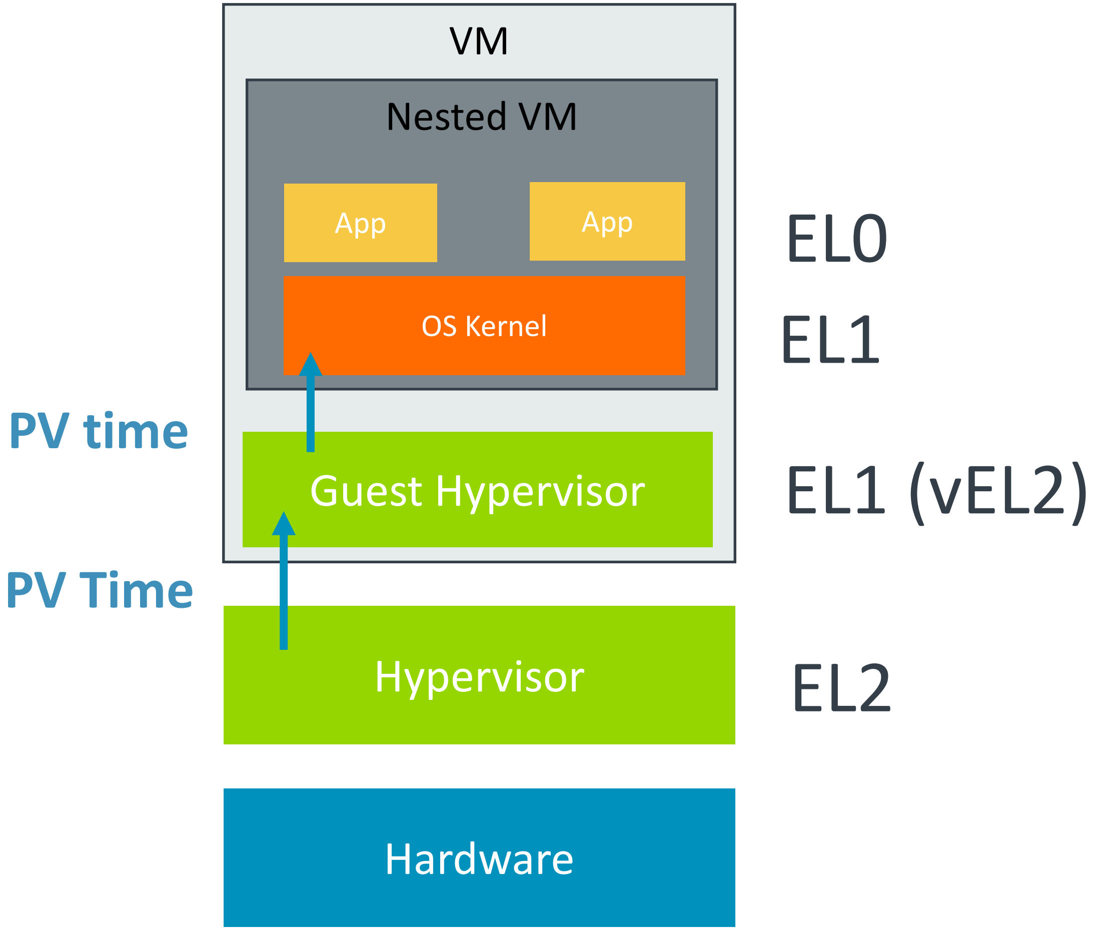
Migration makes things harder…

Combining LPT at several levels, and scaling that to Fpv, is even harder…

Nested Virtualization: Oh no, more turtles…

BETA: WORK IN PROGRESS!

We now have two instances of PV time



Nested Virtualization: Oh no, more turtles...

BETA: WORK IN PROGRESS!

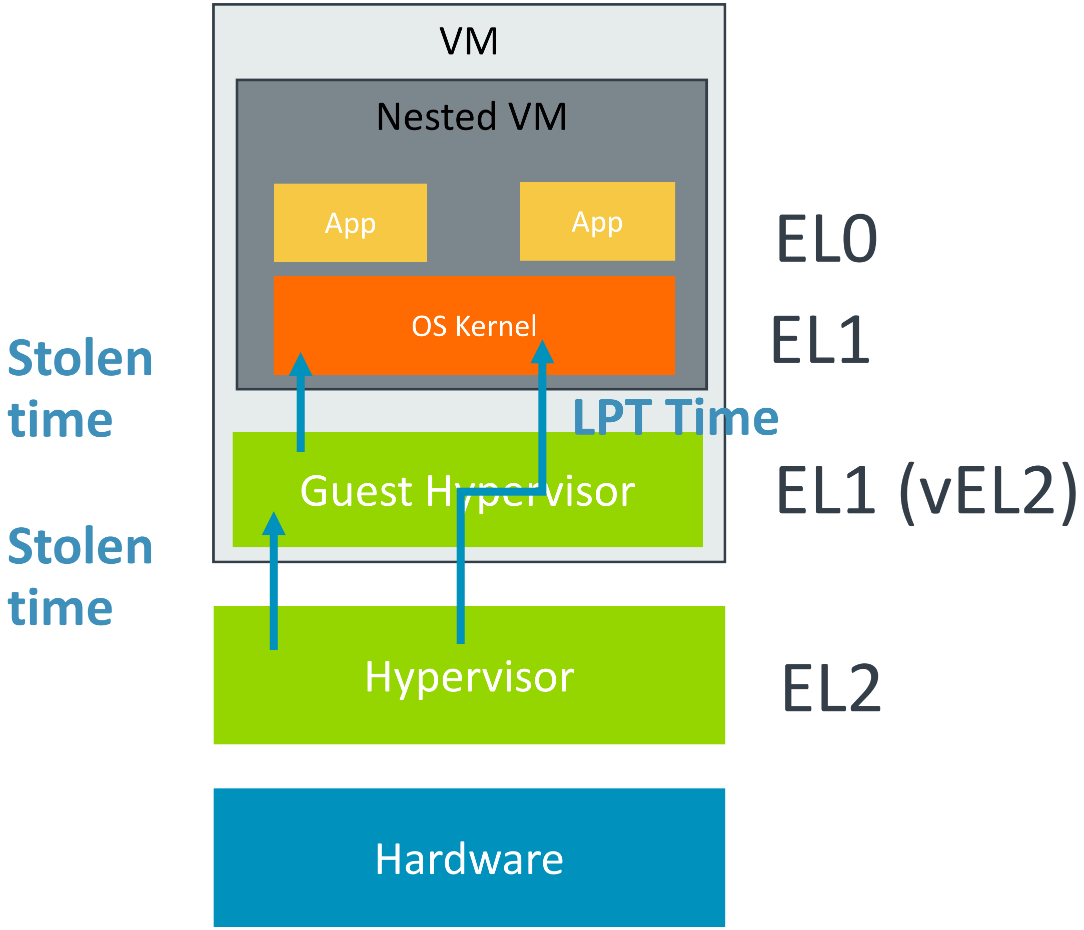
We now have two instances of PV time

We can share the host LPT Time structure

Guest guest OS directly sees shift and multiplier

Though not the stolen time structure

The guest hypervisor must present its own stolen time to the nested VMs
(potentially taking host stolen time into account)



Nested Virtualization: Combining Offsets

BETA: WORK IN PROGRESS!

CNTVOFF_EL2 controlled by host hypervisor and expresses paused time to VM (guest hypervisor)

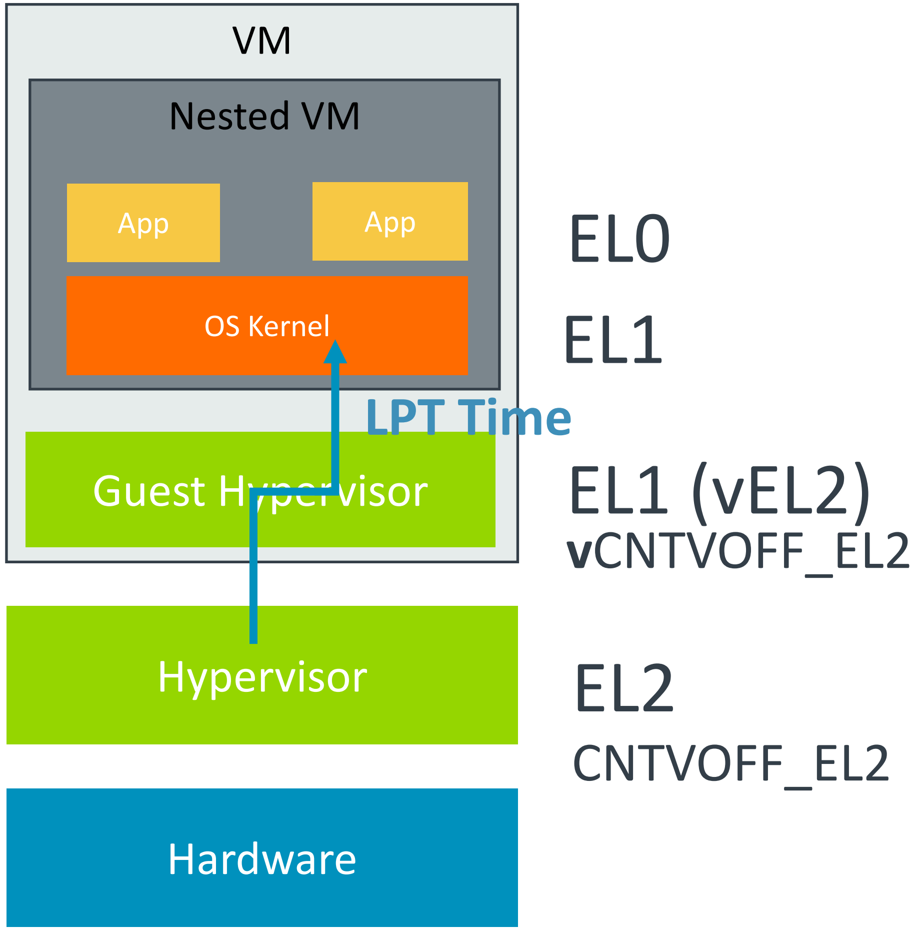
vCNTVOFF_EL2 (virtual offset) is set by guest hypervisor and expresses paused time to nested VM (guest guest OS).

When running guest hypervisor:

CNTVOFF_EL2 = paused

When running guest guest OS:

CNTVOFF_EL2 = paused + vCNTVOFF_EL2



arm

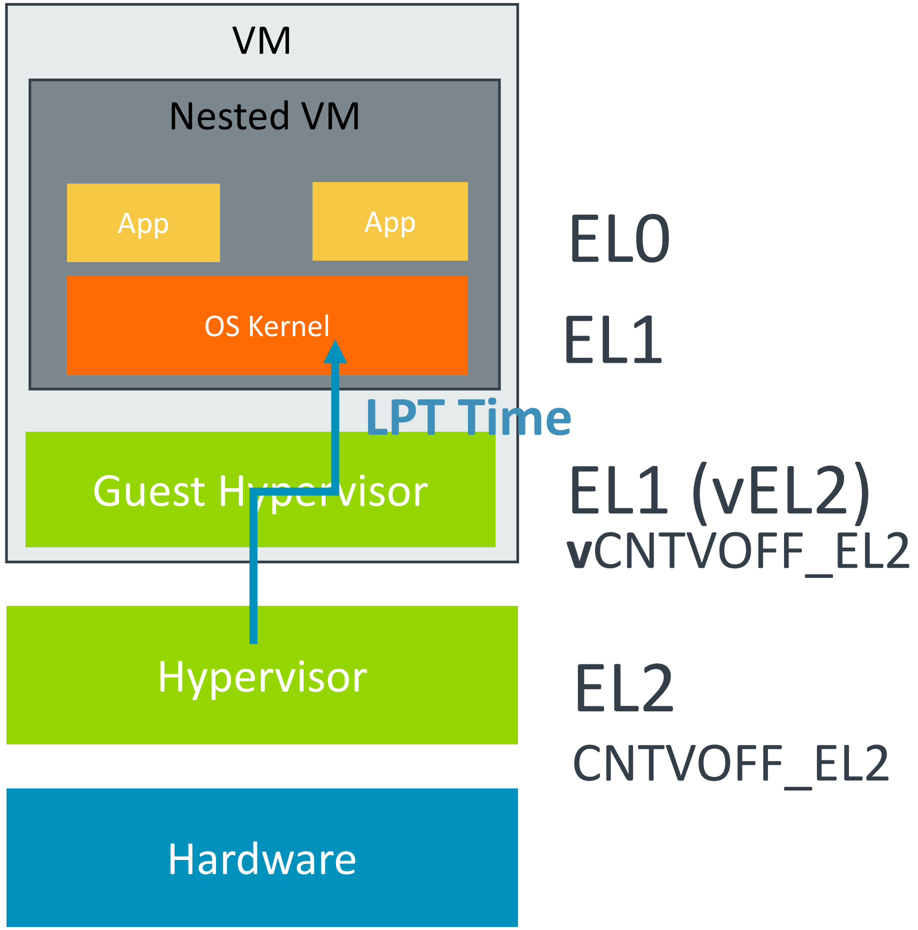
Nested Virtualization: Combining Offsets

BETA: WORK IN PROGRESS!

New semantics for CNTVOFF_EL2 for guest hypervisors:

vCNTVOFF_EL2, when using PV Time, is offset from **virtual** counter, CNTVCT_EL0.

Host hypervisor can scale combined value no migrations with different frequencies.



Conclusions

The Arm architecture has some support for virtual time.

But we still need PV Time to address migration and stolen time.

New (BETA!) specification: Paravirtualized Time for Arm-based Systems

http://https://developer.arm.com/docs/den0057/a

Nested virtualization hurts!

Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!
감사합니다
धन्यवाद