

October 2020, KVM Forum

Virtual Topology for VMs: Friend or Foe ?

Dario Faggioli <dfaggioli@suse.com>

Software Engineer - Virtualization Specialist, SUSE

Some Words of Wisdom...

“In theory, there is no difference between theory and practice. But, in practice, there is.” [Jan L. A. van de Snepscheut](#)

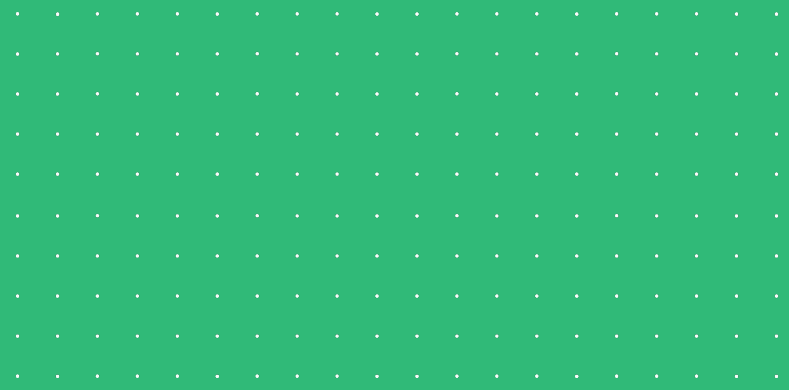
Alternative formulations:

- “It is virtually possible to provide the VM with a topology that matches the physical one. But be mindful of all the potential pitfalls”
- “Can be very good... But also a total mess! Always check everything not two but three 3 times, and keep your finger crossed like there is no tomorrow!”



"Wisdom" by zigazou76 is licensed under CC BY 2.0

About This Talk



“Agenda” (I)

This talk will be about:

1. This function of the Linux kernel scheduler:
`try_to_wake_up()`

```
static int
try_to_wake_up(struct task_struct *p,
               unsigned int state, int wake_flags)
{
    preempt_disable();

    raw_spin_lock_irqsave(&p->pi_lock, flags);

    trace_sched_waking(p);

    p->state = TASK_WAKING;

    cpu = select_task_rq(p, p->wake_cpu,
                        SD_BALANCE_WAKE);

    ttwu_queue(p, cpu, wake_flags);

    raw_spin_unlock_irqrestore(&p->pi_lock, flags);

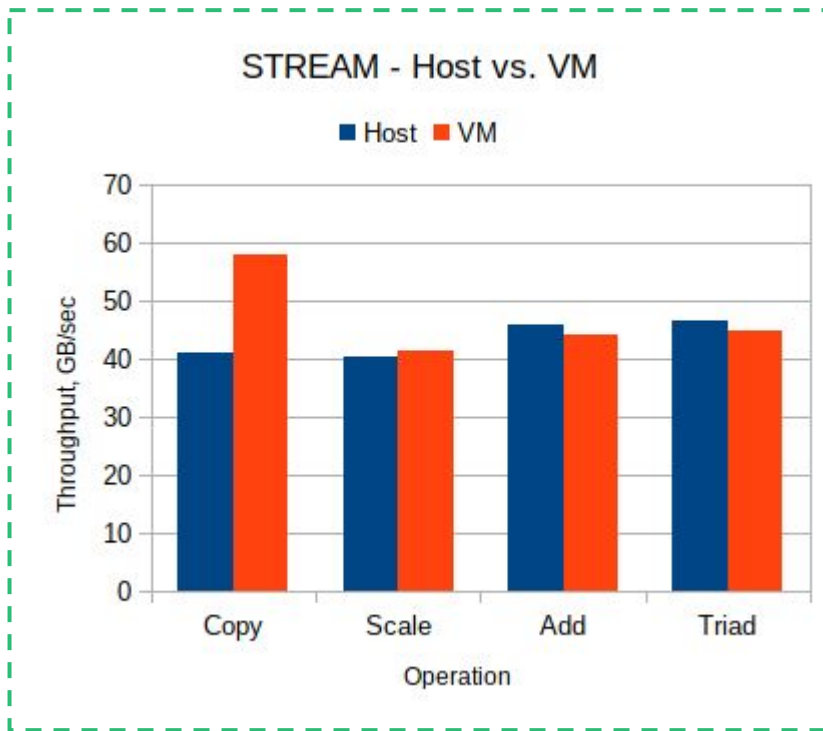
    preempt_enable();

    return;
}
```

“Agenda” (II)

This talk will be about (well, sort of..):

1. This function of the Linux kernel scheduler:
`try_to_wake_up()`
2. This graph



“Agenda” (III)

This talk will be about (well, sort of..):

1. This function of the Linux kernel scheduler:
`try_to_wake_up()`
2. This graph
3. These messages and vulnerability status report seen inside a Virtual Machine

```
# dmesg
[...]
L1TF: System has more than MAX_PA/2 memory.
      L1TF mitigation not effective
[...]

# cd /sys/devices/system/cpu/vulnerabilities
# grep -H . *
Itlb_multihit: Not affected
l1tf: Vulnerable
mds: Clear CPU buffers; SMT Host state unknown
meltdown: PTI
spec_store_bypass: Speculative Store Bypass disabled
via prctl and seccomp
spectre_v1: usercopy/swapgs barriers and __user
pointer sanitization
spectre_v2: Full generic retpoline, IBPB: conditional,
IBRS_FW, STIBP: conditional, RSB filling
Srbds: Not affected
tsx_async_abort: Clear CPU buffers; SMT vulnerable
```

Background Concepts



"Prerequisite to pie" by cvanstane is licensed under CC BY 2.0

Topology

- Topology in Mathematics

“Topology (from the Greek words **τόπος**, 'place, location', and **λόγος**, 'study') is concerned with the properties of a geometric object that are preserved under continuous deformations, such as stretching, twisting, crumpling and bending, but not tearing or gluing.”

- Topology in Electrical Circuits

“The topology of an electronic circuit is the form taken by the network of interconnections of the circuit components. [...]. Topology is not concerned with the physical layout of components in a circuit, nor with their positions on a circuit diagram; similarly to the mathematics concept of topology, it is only concerned with what connections exist between the components.”



"Topology" by bmeabroad is licensed under CC BY-NC-SA 2.0

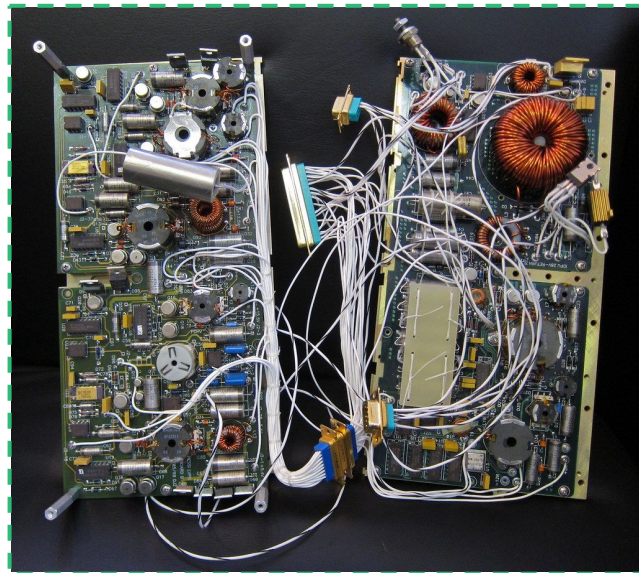
Physical Topology

CPUs, caches, memory, IO devices



Topology = Organization

- Threads, cores, dies, packages, sockets
- L1, L2, L3 cache hierarchy
- NUMA nodes
- IO buses & bridges



"Opening a Satellite" by jurvetson is licensed under CC BY 2.0

Physical Topology

Inspecting the physical (CPU & memory) topology:

- `lscpu`

```
# lscpu
Architecture:          x86_64
CPU op-mode(s):       32-bit, 64-bit
Byte Order:           Little Endian
Address sizes:        46 bits physical, 48 bits
virtual
CPU(s):               224
On-line CPU(s) list: 0-223
Thread(s) per core:   2
Core(s) per socket:  28
Socket(s):            4
NUMA node(s):        4
[...]
L1d cache:           32K
L1i cache:           32K
L2 cache:            1024K
L3 cache:            39424K
NUMA node0 CPU(s):   0-27,112-139
NUMA node1 CPU(s):   28-55,140-167
NUMA node2 CPU(s):   56-83,168-195
NUMA node3 CPU(s):   84-111,196-223
```

Physical Topology

Inspecting the physical (CPU & memory) topology:

- `lscpu`
- `numactl --hardware`

```
# numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
              15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
              30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
              45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
              60 61 62 63 64 65 66 67 68 69 70 71
node 0 size: 31849 MB
node 0 free: 24587 MB
node 1 cpus: 24 25 26 27 28 29 30 31 32 33 34 35
              36 37 38 39 40 41 42 43 44 45 46 47
              48 49 50 51 52 53 54 55 56 57 58 59 60 61
              62 63 64 65 66 67 68 69 70 71
node 1 size: 32240 MB
node 1 free: 31640 MB
node distances:
node  0  1
 0:  10  21
 1:  21  10
```

Physical Topology

Inspecting the physical (CPU & memory) topology:

- `lscpu`
- `numactl --hardware`
- Linux `sysfs` interfaces

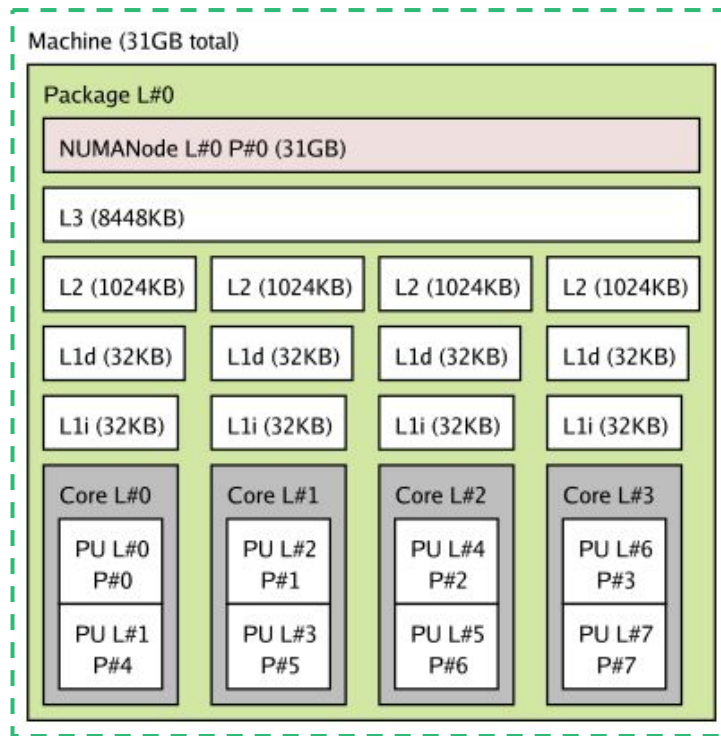
```
# cd /sys/devices/system/cpu
# cat cpu0/topology/package_cpus_list
0-27,112-139
$ cat cpu0/topology/core_siblings_list
0-27,112-139
$ cat cpu0/topology/core_cpus_list
0,112
$ cat cpu/cpu0/topology/thread_siblings_list
0,112

# cat cpu0/cache/index0/level
1
# cat cpu0/cache/index1/type
Instruction
# cat cpu0/cache/index2/ways_of_associativity
16
# cat cpu0/cache/index3/type
Unified
# cat cpu0/cache/index3/shared_cpu_list
0-27,112-139
```

Physical Topology

Inspecting the physical (CPU & memory) topology:

- `lscpu`
- `numactl --hardware`
- Linux `sysfs` interfaces
- `lstopo`, from [Portable Hardware Locality](#)



Topology of a Virtual Machine

Try these same tools when inside a VM:

- `lscpu`
- `numactl --hardware`
- Linux `sysfs` interfaces
- `lstopo`, from [Portable Hardware Locality](#)

⇒ They all work!

- And give similar output than on Bare Metal

⇒ You're looking at the Virtual Topology



"Virtual Reality Demonstrations" by
UTKnightCenter is licensed under CC BY 2.0

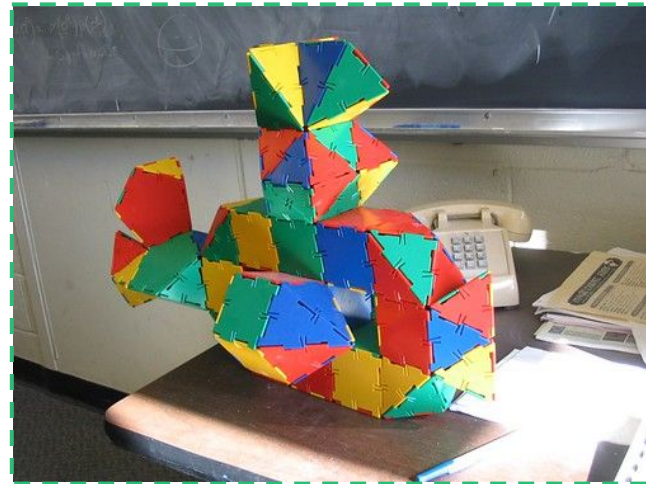
Virtual Topology

It's created for the VM by the hypervisor and the tools (e.g., KVM, QEMU and Libvirt)

- Two VMs on the same hardware, can have different virtual topologies
- A (across reboot) change its topology

How does this work?

- The user specifies what he/she wants
 - **Sockets, Core x Socket, Threads x Core**
 - **NUMA nodes**
 - **NUMA distances**
 - ...
- The tools prepare the virtual hardware (virtual ACPI tables, CPUID, ...)
- The guest kernel query the topology, like it were on a baremetal system
- Result is the requested Virtual Topology

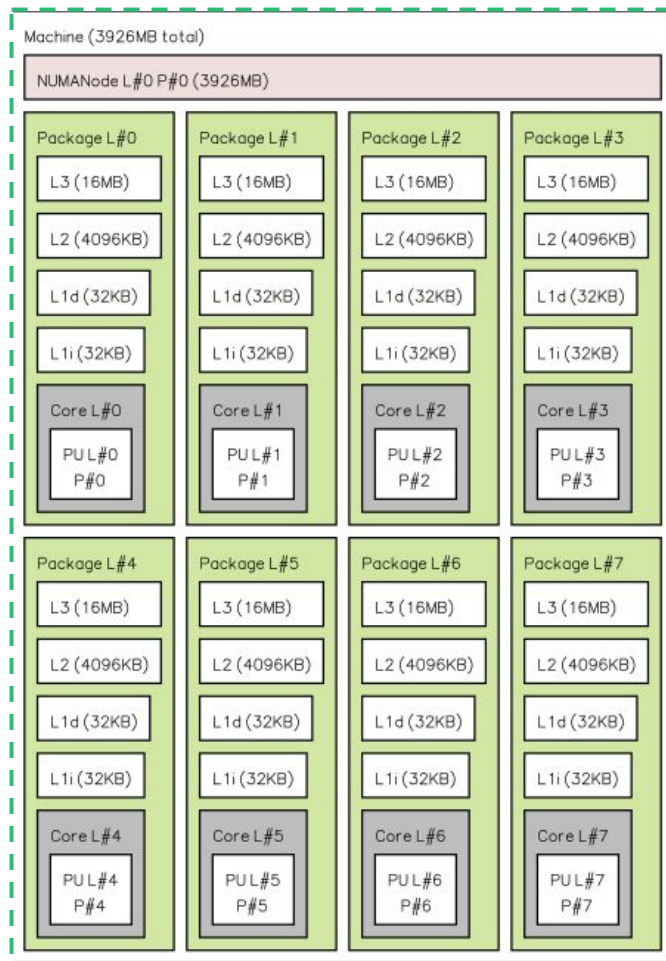


"Euler's battlecruiser 1" by shonk is licensed under CC BY 2.0

Virtual Topology: Examples

- Default Topology

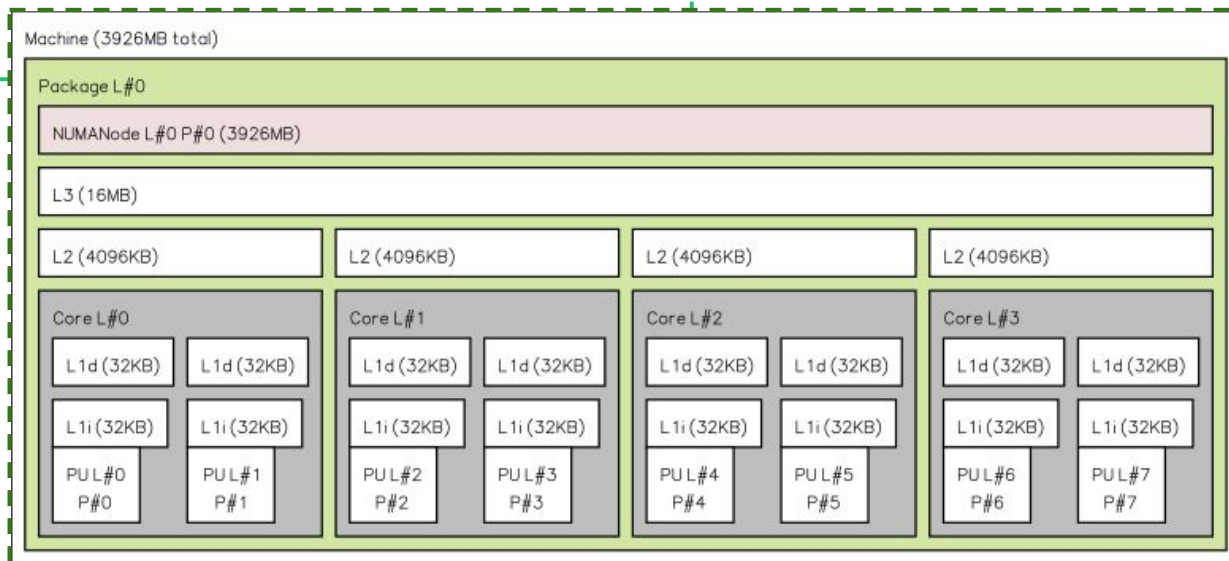
```
<vcpu placement="static">8</vcpu>  
<cpu mode="host-passthrough" check="none">  
  <topology sockets="8" dies="1" cores="1" threads="1"/>  
</cpu>
```



Virtual Topology: Examples

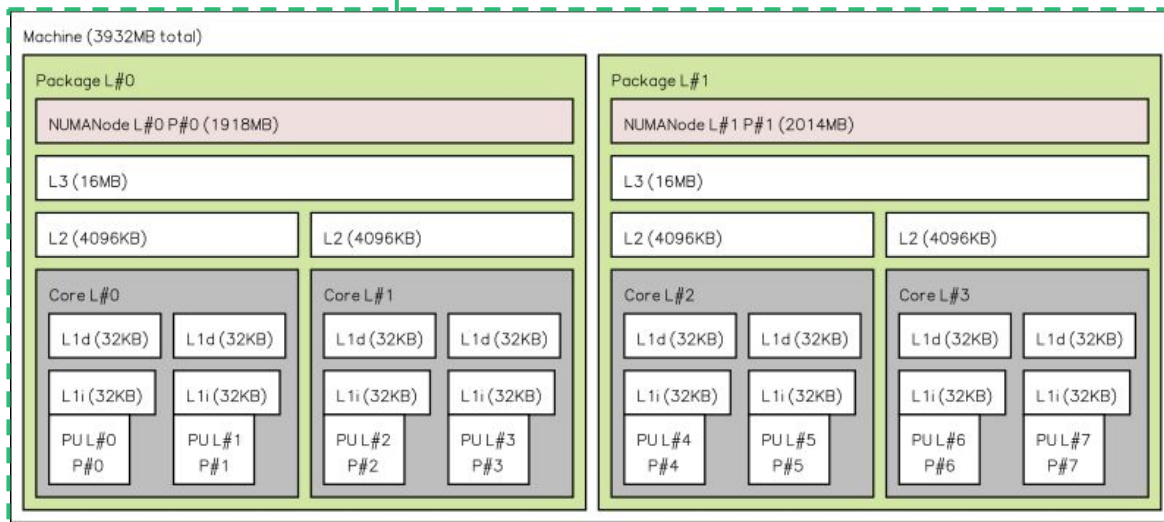
- With Cores and Threads

```
<vcpu placement="static">8</vcpu>  
<cpu mode="host-passthrough" check="none">  
  <topology sockets="1" dies="1" cores="4" threads="2"/>  
</cpu>
```



Virtual Topology: Examples

```
<vcpu placement="static">8</vcpu>
<cpu mode="host-passthrough" check="none">
  <topology sockets="1" dies="1" cores="4" threads="2"/>
  <numa>
    <cell id='0' cpus='0-3'
      memory='2048' unit='MiB'>
      <distances>
        <sibling id='0' value='10'/>
        <sibling id='1' value='21'/>
      </distances>
    </cell>
    <cell id='1' cpus='4-7'
      memory='2048' unit='MiB'>
      <distances>
        <sibling id='0' value='21'/>
        <sibling id='1' value='10'/>
      </distances>
    </cell>
  </numa>
</cpu>
```



Topology & Performance

Huge benefits come from a “Topology Aware” OS

The scheduler is a typical example:

- Don't run tasks on CPUs that share computational resources, if possible (SMT aware scheduling)
 - ⇒ **improves performance**
- Run tasks on the NUMA node where the memory they use is (NUMA aware scheduling)
 - ⇒ **improves scalability**
- Task load balancing done hierarchically, through scheduling domains mapped on the topology
 - ⇒ **improves scalability**
- Pack tasks on CPUs sharing power domains, so other CPUs can be put into low power state (Power aware scheduling)
 - ⇒ **improves efficiency**



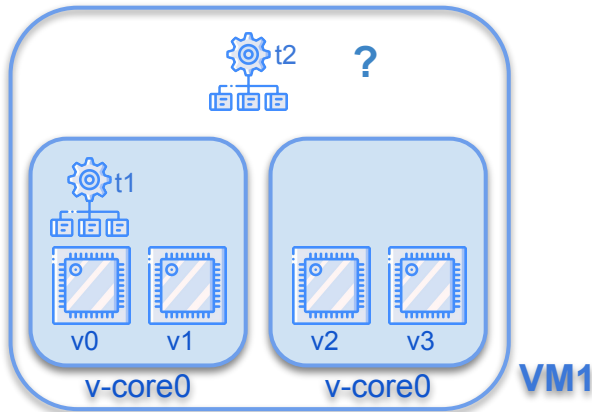
"Odd Future Performing" by Incase. is licensed under CC BY 2.0

Did I hear “Performance” ?

Virtual Topology & Performance

The OS in the VM is aware of the Virtual Topology

- SMT aware guest scheduler, in VM1:
 - Virtual CPU v0 from virtual core v-core0 is busy running task t1
 - Where should I run t2, in order to **avoid** having two physical threads busy at the same time, if possible?



"Question" by ryanvanetten is licensed under CC BY-SA 2.0

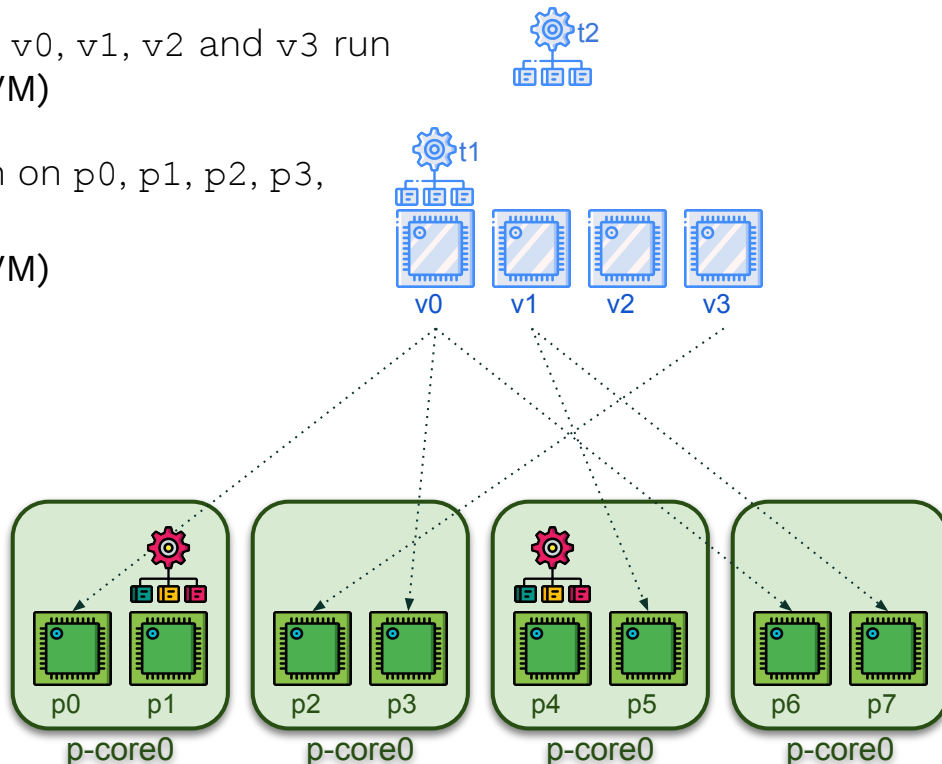
SUSE

Virtual Topology & Performance

- Depends on what physical CPUs v_0 , v_1 , v_2 and v_3 run
 - No way to tell (from the VM)
 - Changes over time
- Depends on what host tasks run on p_0 , p_1 , p_2 , p_3 , p_4 , p_5 and p_6
 - No way to tell (from the VM)
 - Changes over time

⇒ Is there even hope !?!

⇒ Isn't it better to just **NOT** define a Virtual Topology ?



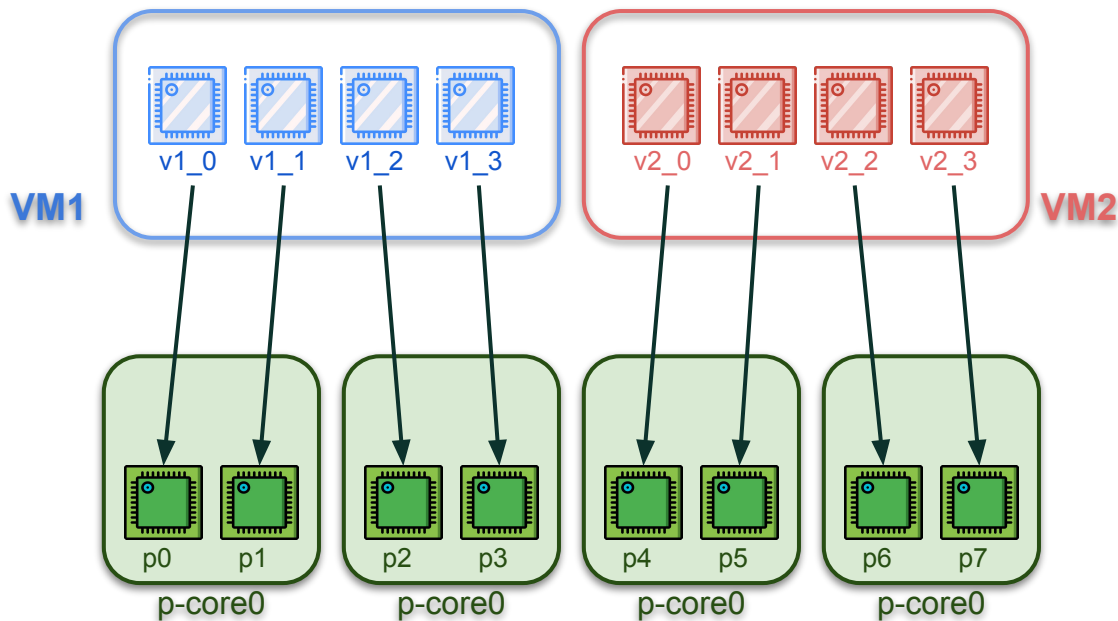
Resource Partitioning

What if, instead, relationship between virtual and physical CPUs were:

- Well defined
- Constant over time (e.g., for the lifetime of the VMs)

I.e.:

- `v1_0` always runs on `p0`
- `v1_1` always runs on `p1`
- `v1_2` always run on `p2`
- `v1_3` always run on `p3`
- `v2_0` always run on `p4`
- `v2_1` always run on `p5`
- `v2_2` always run on `p6`
- `v2_3` always run on `p7`



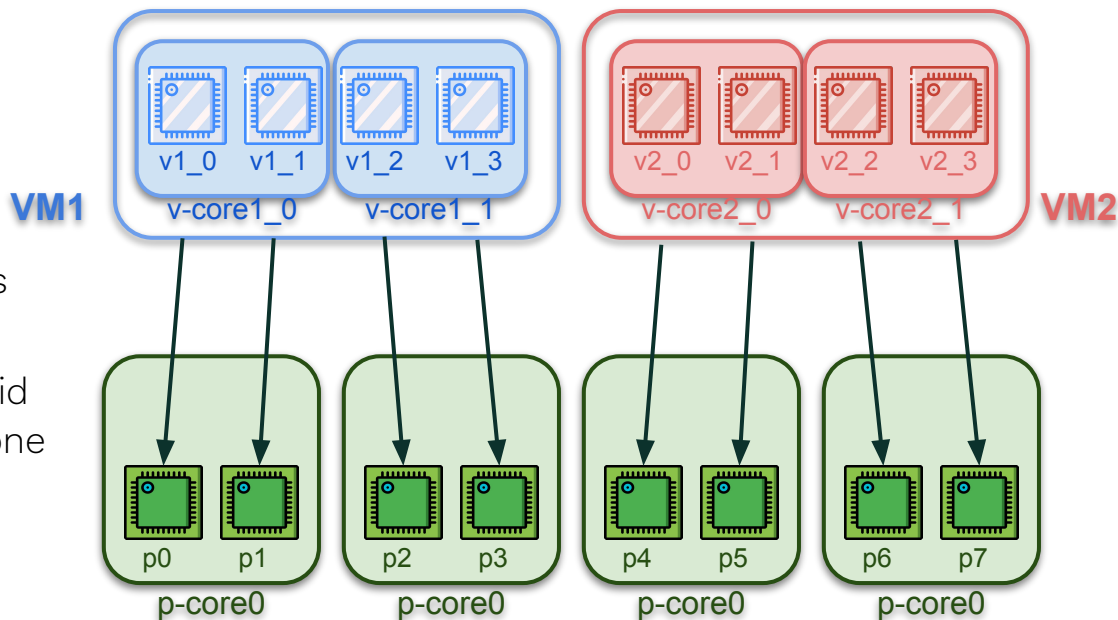
Resource Partitioning

What if, instead, relationship between virtual and physical CPUs were:

- Well defined
- Constant over time (e.g., for the lifetime of the VMs)

Now a Virtual Topology that matches the underlying Physical Topology makes sense

- In-VM scheduler decisions are valid
- They're ~ like they'd have been done on the host



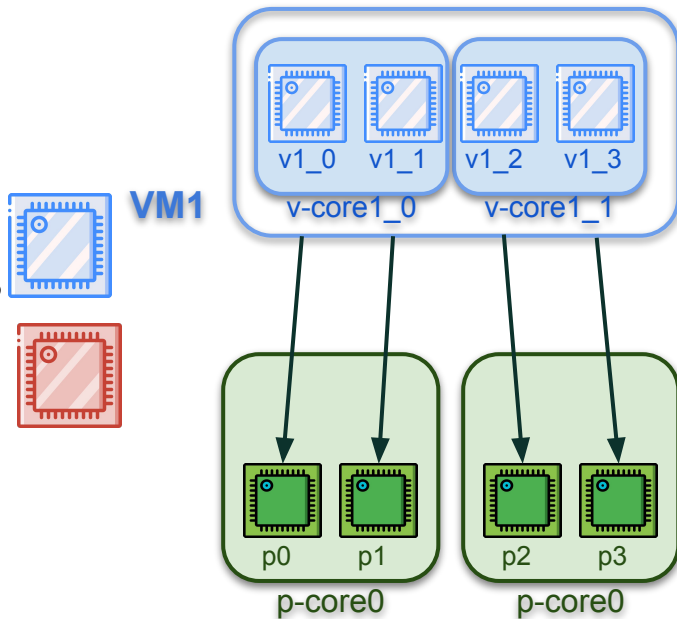
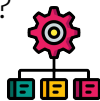
Dedicated Resource Partitioning

Resource Partitioning:

- v1_0 runs only on p0
- v1_1 runs only on p1
- v1_2 runs only on p2
- v1_3 runs only on p3

What about p0, p1, p2, p3 ?

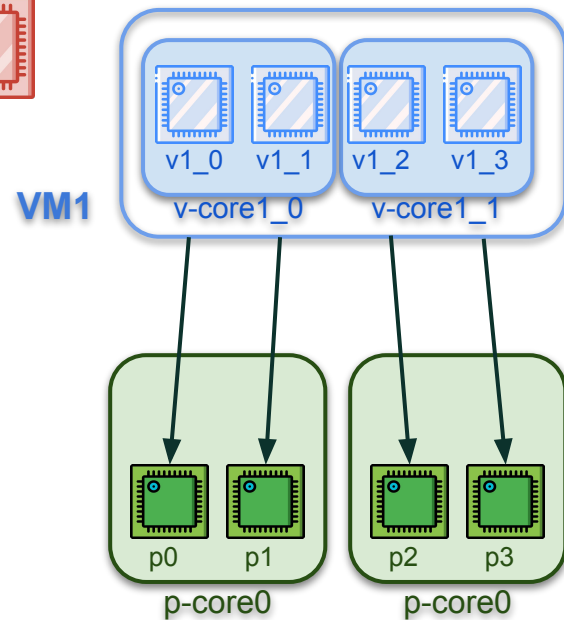
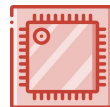
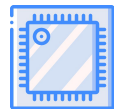
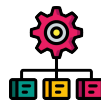
1. Do they also only run v1_0, v1_1, v1_2 and v1_3 ?
Or can also other vCPUs from other VMs run there?
2. Can they run the vCPUs, but also host tasks?



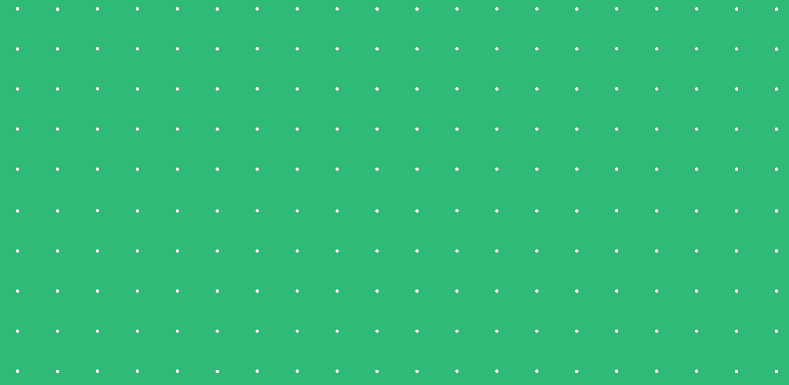
Dedicated Resource Partitioning

What about p0, p1, p2, p3 ?

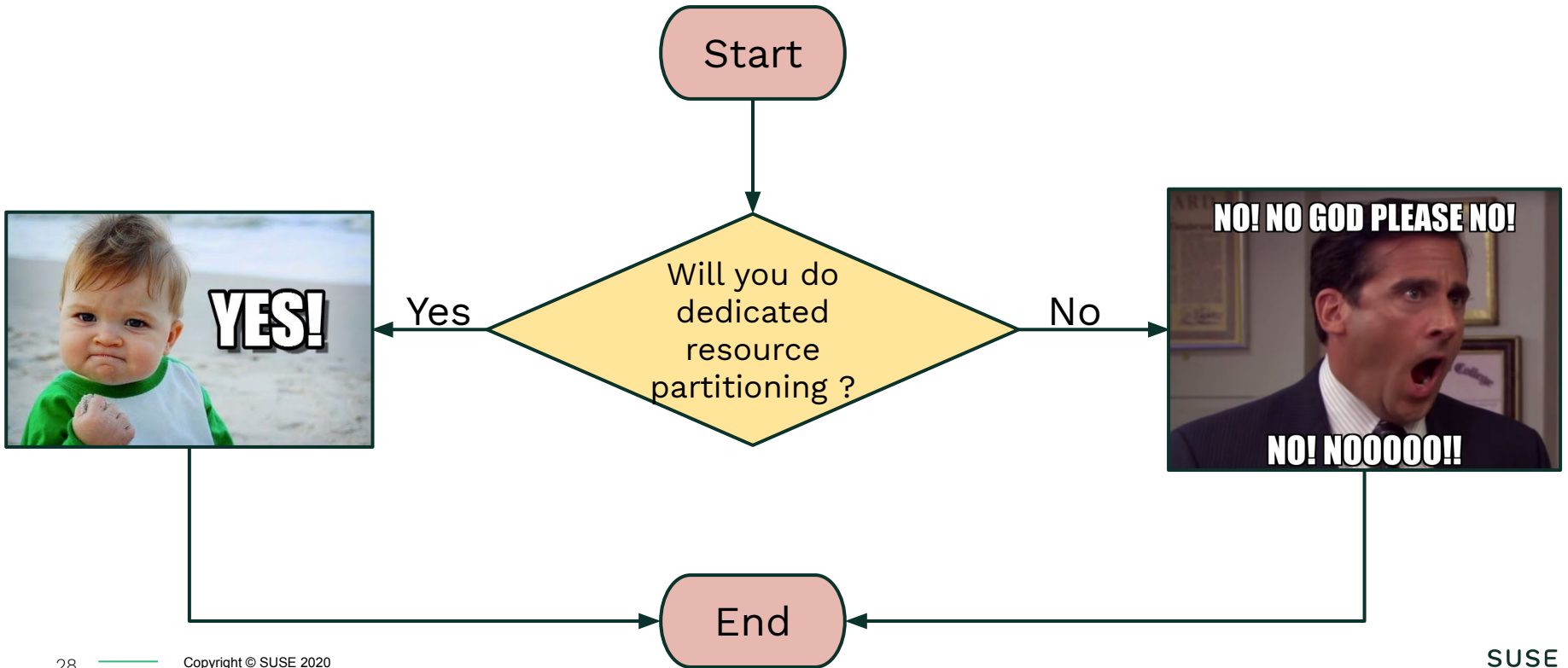
1. Do they also only run v1_0, v1_1, v1_2 and v1_3 ?
Or can also other vCPUs from other VMs run there?
 - If only vCPUs of VM1 can run on them, we are doing dedicated resource partitioning for our VMs
2. Can they run the vCPUs, but also host tasks?
 - If yes, VM1 will incur in some interference. It is probably still fine and we can consider this ~ dedicated partitioning, If host load is low and interference is rare



Virtual Topology: State of the Union



Shall I Define a Virtual Topology for my VM ?



Virtual CPU & Memory Pinning

Resource Partitioning is enacted via vCPU pinning

- Can be done in Libvirt
- For each vCPU, we specify on which Physical CPU(s) it can run on
- We specify on which physical NUMA node(s) of the the memory should be allocated on

```
<vcpu placement="static">8</vcpu>
<cputune>
  <vcpupin vcpu="0" cpuset="36"/>
  <vcpupin vcpu="1" cpuset="84"/>
  <vcpupin vcpu="2" cpuset="37"/>
  <vcpupin vcpu="3" cpuset="85"/>
</cputune>
<numatune>
  <memory mode="strict" nodeset="1"/>
</numatune>
```



"pins" by hydropeek is licensed under CC BY 2.0

Virtual Topology & CPU Pinning

- Establishing a 1 to 1 mapping between Real and Virtual Topology and doing the pinning can be cumbersome
- QEMU follow a specific order
 - E.g.: 2 cores, 2 threads:
 - vCPU 0 and 1 → virtual core 1, as threads
 - vCPU 2 and 3 → virtual core 2, as threads
- Physical CPUs
 - IDs can be sparse

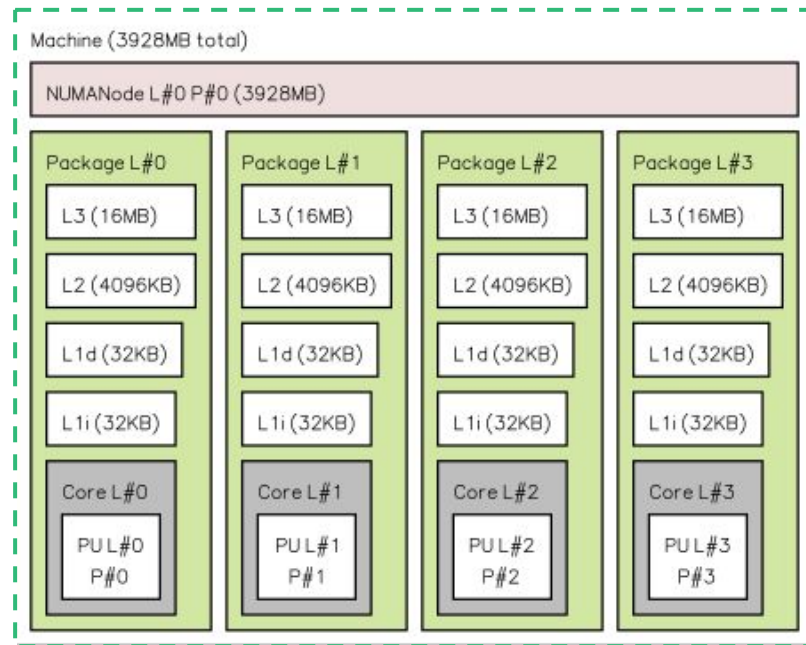
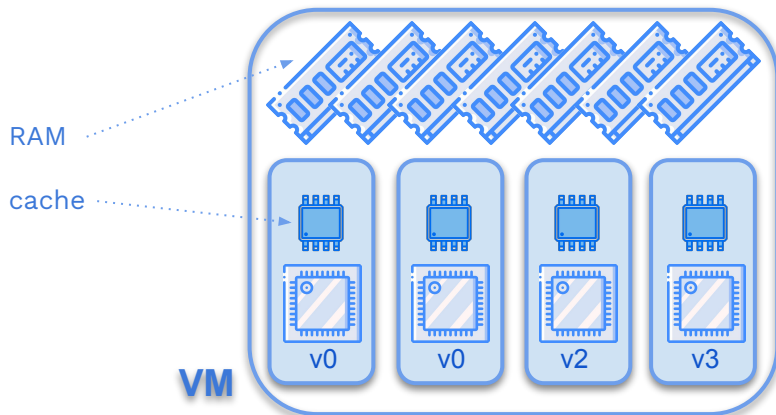
```
NUMA node0 CPU(s): 0, 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120
NUMA node1 CPU(s): 2, 10, 18, 26, 34, 42, 50, 58, 66, 74, 82, 90, 98, 106, 114, 122
NUMA node2 CPU(s): 4, 12, 20, 28, 36, 44, 52, 60, 68, 76, 84, 92, 100, 108, 116, 124
NUMA node3 CPU(s): 6, 14, 22, 30, 38, 46, 54, 62, 70, 78, 86, 94, 102, 110, 118, 126
NUMA node4 CPU(s): 1, 9, 17, 25, 33, 41, 49, 57, 65, 73, 81, 89, 97, 105, 113, 121
NUMA node5 CPU(s): 3, 11, 19, 27, 35, 43, 51, 59, 67, 75, 83, 91, 99, 107, 115, 123
NUMA node6 CPU(s): 5, 13, 21, 29, 37, 45, 53, 61, 69, 77, 85, 93, 101, 109, 117, 125
NUMA node7 CPU(s): 7, 15, 23, 31, 39, 47, 55, 63, 71, 79, 87, 95, 103, 111, 119, 127
```

```
<cpupin vcpu='0' cpuset='0' />
<cpupin vcpu='1' cpuset='64' />
<cpupin vcpu='2' cpuset='8' />
<cpupin vcpu='3' cpuset='72' />
<cpupin vcpu='4' cpuset='16' />
<cpupin vcpu='5' cpuset='80' />
<cpupin vcpu='6' cpuset='24' />
<cpupin vcpu='7' cpuset='88' />
<cpupin vcpu='8' cpuset='32' />
<cpupin vcpu='9' cpuset='96' />
<cpupin vcpu='10' cpuset='40' />
<cpupin vcpu='11' cpuset='104' />
<cpupin vcpu='12' cpuset='48' />
<cpupin vcpu='13' cpuset='112' />
<cpupin vcpu='14' cpuset='56' />
<cpupin vcpu='15' cpuset='120' />
<cpupin vcpu='16' cpuset='2' />
<cpupin vcpu='17' cpuset='66' />
[...]
```

Default Virtual Topology

E.g., for a 4 vCPUs VM:

- All vCPUs as sockets \Rightarrow flat
 - No relationship / dependencies between vCPUs
- No sharing, not even caches
 - Not so common in hardware



`try_to_wake_up()`
In some details...



Waking Up Tasks

In the Linux kernel, `try_to_wake_up()` is called when a task that was blocked or sleeping, wants to run again

1. The wake-up of the task (E.g., `t1`) happens on a CPU, the wakeup CPU (e.g., `p0`)
2. The task needs to be put in a runqueue, the target runqueue (E.g., `p1_rq` or `p2_rq`)
3. The target CPU is informed about the new task
 - a. If the target CPU is idle, the task runs
 - b. If the target CPU is busy, it checks for preemption

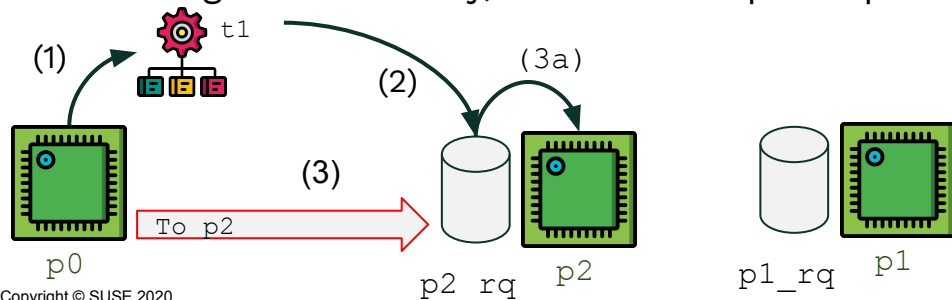


"Wakeup, smurf!" by Willem Cantor is licensed under CC BY-SA 2.0

Waking Up Tasks

In the Linux kernel, `try_to_wake_up()` is called when a task that was blocked or sleeping, wants to run again

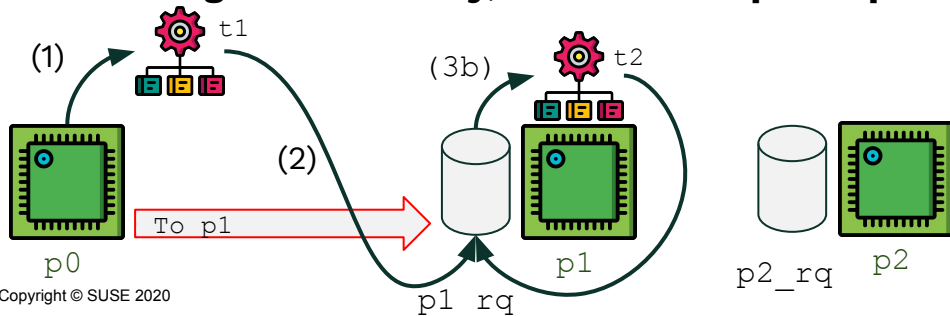
1. The wake-up of the task (E.g., `t1`) happens on a CPU, the wakeup CPU (e.g., `p0`)
2. The task needs to be put in a runqueue, the target runqueue (E.g., `p1_rq` or `p2_rq`)
3. The target CPU is informed about the new task
 - a. **If the target CPU is idle, the task runs**
 - b. If the target CPU is busy, it checks for preemption



Waking Up Tasks

In the Linux kernel, `try_to_wake_up()` is called when a task that was blocked or sleeping, wants to run again

1. The wake-up of the task (E.g., `t1`) happens on a CPU, the wakeup CPU (e.g., `p0`)
2. The task needs to be put in a runqueue, the target runqueue (E.g., `p1_rq` or `p2_rq`)
3. The target CPU is informed about the new task
 - a. If the target CPU is idle, the task runs
 - b. **If the target CPU is busy, it checks for preemption**



The Topology of a Wake-up

`try_to_wake_up()` deals with the topology:

- `p0` (wakeup CPU) and `p2` (target CPU), share an L3 cache; `p2` is idle
- `p2`'s runqueue structure belongs (and will likely be) in the L3 cache

```
trace-cmd record function_graph -g try_to_wake_up -g *resched* -e sched_waking -e reschedule_entry  
-e sched_wake_idle_without_ipi
```

```
[p0] try_to_wake_up()
```

```
[p0] comm=t1 prio=120 target_cpu=p2
```

```
[p0] ttwu_queue(p, cpu)
```

```
[p0] if (cpus_share_cache(p0, p2))
```

```
[p0] ttwu_do_activate()
```

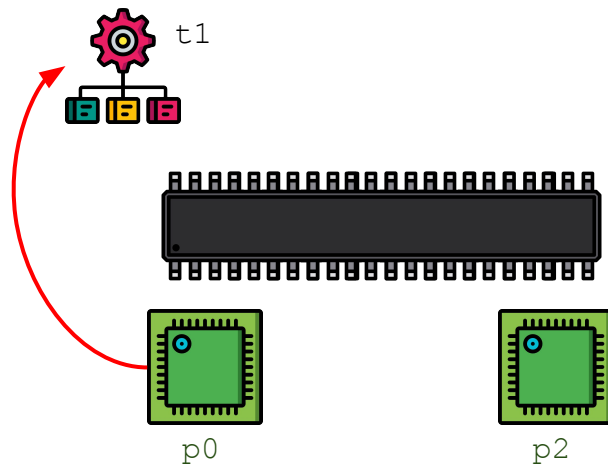
```
[p0] enqueue_task_fair();
```

```
[p0] ttwu_do_wakeup()
```

```
[p0] check_preempt_curr()
```

```
[p0] resched_curr()
```

```
[p0] sched_wake_idle_without_ipi: cpu=p2
```



The Topology of a Wake-up

`try_to_wake_up()` deals with the topology:

- `p0` (wakeup CPU) and `p2` (target CPU), share an L3 cache; `p2` is idle
- `p2`'s runqueue structure belongs (and will likely be) in the L3 cache

```
trace-cmd record function_graph -g try_to_wake_up -g *resched* -e sched_waking -e reschedule_entry  
-e sched_wake_idle_without_ipi
```

```
[p0] try_to_wake_up()
```

```
[p0] comm=t1 prio=120 target_cpu=p2
```

```
[p0] ttwu_queue(p, cpu)
```

```
[p0] if (cpus_share_cache(p0, p2))
```

```
[p0] ttwu_do_activate()
```

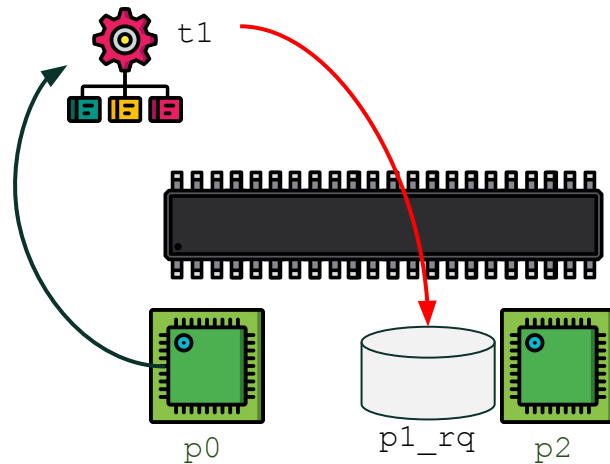
```
[p0] enqueue_task_fair();
```

```
[p0] ttwu_do_wakeup()
```

```
[p0] check_preempt_curr()
```

```
[p0] resched_curr()
```

```
[p0] sched_wake_idle_without_ipi: cpu=p2
```



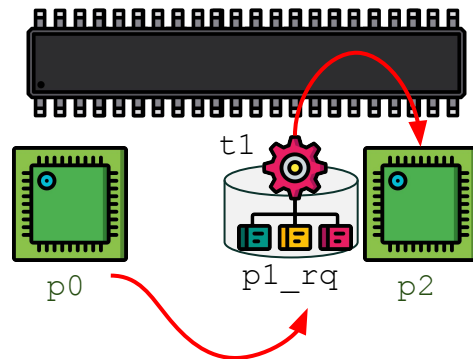
The Topology of a Wake-up

`try_to_wake_up()` deals with the topology:

- `p0` (wakeup CPU) and `p2` (target CPU), share an L3 cache; `p2` is idle
- `p2`'s runqueue structure belongs (and will likely be) in the L3 cache

```
trace-cmd record function_graph -g try_to_wake_up -g *resched* -e sched_waking -e reschedule_entry  
-e sched_wake_idle_without_ipi
```

```
[p0] try_to_wake_up()  
[p0] comm=t1 prio=120 target_cpu=p2  
[p0]   ttwu_queue(p, cpu)  
[p0]     if (cpus_share_cache(p0, p2))  
[p0]       ttwu_do_activate()  
[p0]         enqueue_task_fair();  
[p0]       ttwu_do_wakeup()  
[p0]         check_preempt_curr()  
[p0]         resched_curr()  
[p0] sched_wake_idle_without_ipi: cpu=p2
```



The Topology of Another Wake-up

`try_to_wake_up()` deals with the topology:

- `p0` (wakeup CPU) and `p3` (target CPU), **do not** share an L3 cache; `p3` is idle
- `p3`'s runqueue structure does not belong in `p0`'s L3 cache

```
trace-cmd record function_graph -g try_to_wake_up -g *resched* -e sched_waking -e reschedule_entry  
-e sched_wake_idle_without_ipi
```

```
[p0] try_to_wake_up()
```

```
[p0] comm=t1 prio=120 target_cpu=p3
```

```
[p0] ttwu_queue(p, cpu)
```

```
[p0] if (cpus_share_cache(p0, p3));
```

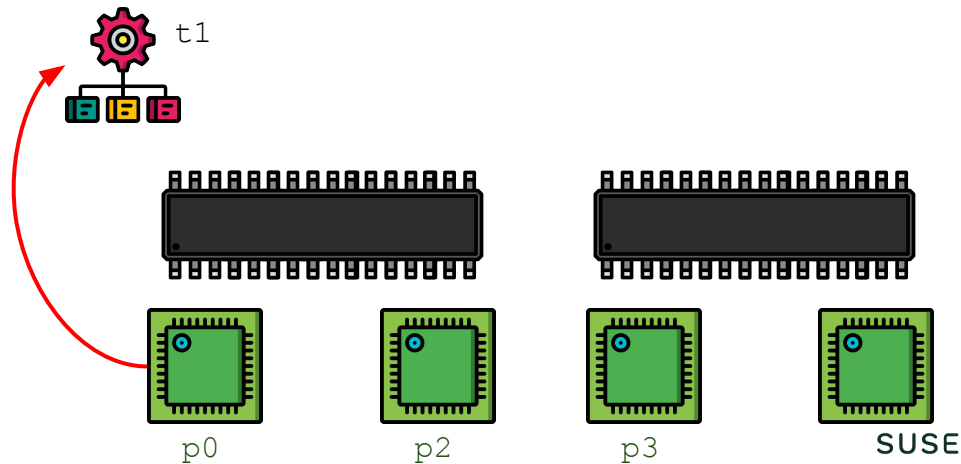
```
[p0] ttwu_queue_wakelist()
```

```
[p3] sched_wake_idle_without_ipi: cpu=p3
```

```
[p3] sched_ttwu_pending()
```

```
[p3] ttwu_do_activate()
```

```
[p3] enqueue_task_fair();
```



The Topology of Another Wake-up

`try_to_wake_up()` deals with the topology:

- `p0` (wakeup CPU) and `p3` (target CPU), **do not** share an L3 cache; `p3` is idle
- `p3`'s runqueue structure does not belong in `p0`'s L3 cache

```
trace-cmd record function_graph -g try_to_wake_up -g *resched* -e sched_waking -e reschedule_entry  
-e sched_wake_idle_without_ipi
```

```
[p0] try_to_wake_up()
```

```
[p0] comm=t1 prio=120 target_cpu=p3
```

```
[p0]   ttwu_queue(p, cpu)
```

```
[p0]   if (cpus_share_cache(p0, p3));
```

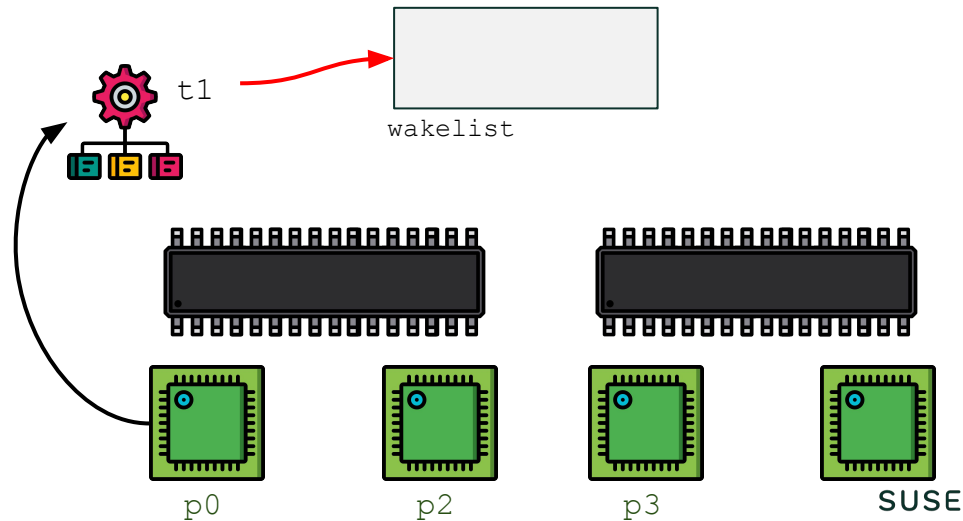
```
[p0]   ttwu_queue_wakelist();
```

```
[p3] sched_wake_idle_without_ipi: cpu=p3
```

```
[p3] sched_ttwu_pending()
```

```
[p3]   ttwu_do_activate()
```

```
[p3]   enqueue_task_fair();
```



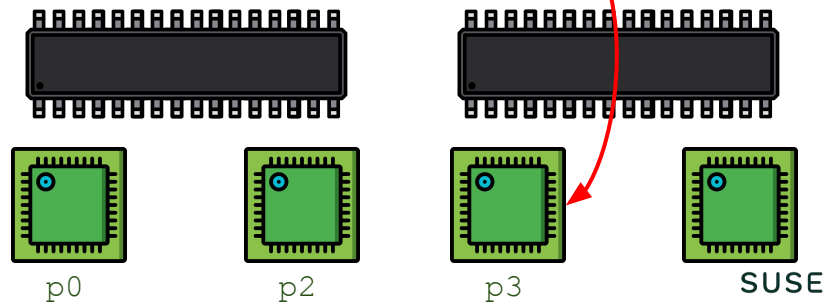
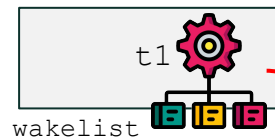
The Topology of Another Wake-up

`try_to_wake_up()` deals with the topology:

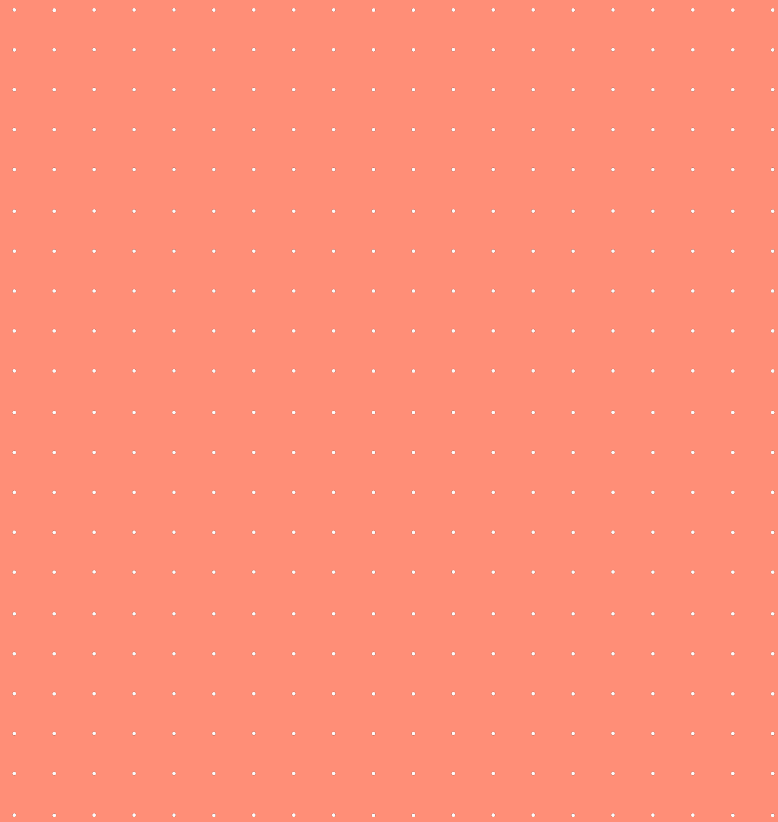
- p0 (wakeup CPU) and p3 (target CPU), **do not** share an L3 cache; p3 is idle
- p3's runqueue structure does not belong in p0's L3 cache

```
trace-cmd record function_graph -g try_to_wake_up -g *resched* -e sched_waking -e reschedule_entry  
-e sched_wake_idle_without_ipi
```

```
[p0] try_to_wake_up()  
[p0] comm=t1 prio=120 target_cpu=p3  
[p0]   ttwu_queue(p, cpu)  
[p0]     if (cpus_share_cache(p0, p3));  
[p0]     ttwu_queue_wakelist();  
[p3] sched_wake_idle_without_ipi: cpu=p3  
[p3] sched_ttwu_pending()  
[p3]   ttwu_do_activate()  
[p3]     enqueue_task_fair();
```



try_to_wake_up(), with a Virtual Topology



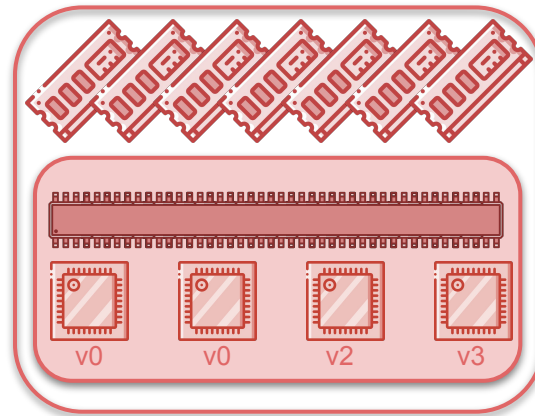
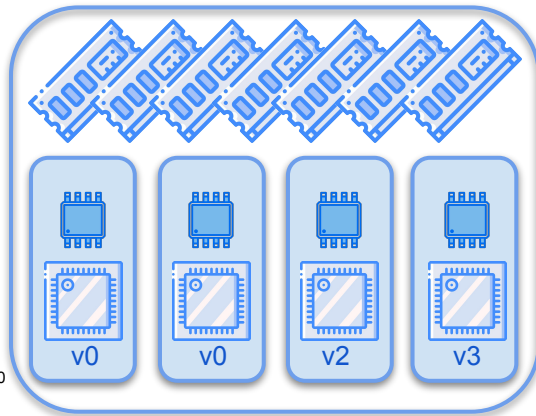
Virtual Topology and Wake-ups in VMs

`try_to_wake_up()` was all about whether the waking and target CPU share caches.

- The same logic is triggered when a task wakes up on a vCPU
- Do virtual CPUs share caches?
 - Before QEMU commit [target-i386: present virtual L3 cache info for vcpus](#) , no!
 - L3 was not part of the virtual topology!
 - Now? Depends on the actual Virtual Topology
 - L3 is there. But is it shared?

All vCPUS as sockets (default topology):

No vCPU share the L3 with any other ones



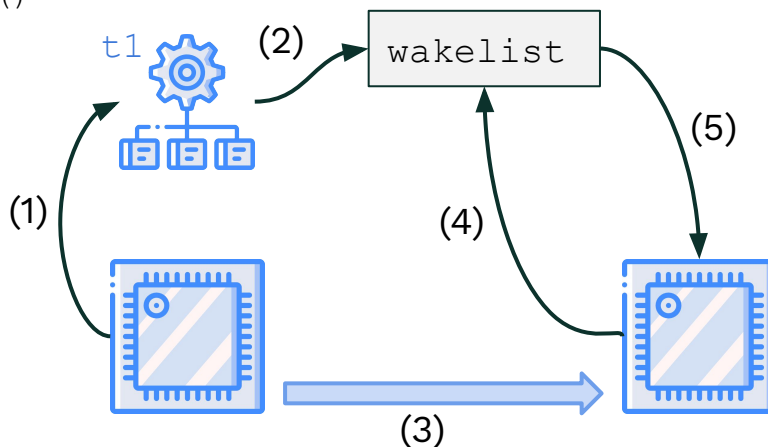
All vCPUS as cores (for instance)
All vCPUs share the same L3

Tracing Wake-ups in VMs

`try_to_wake_up()` in action in a VM:

- p0 (wakeup CPU) and p2 (target CPU), do not share an L3 cache (they never do, by default!)
- p2 is idle

```
[p0] try_to_wake_up()
(1) [p0] sched_waking: comm=t1 prio=120 target_cpu=p2
(2) [p0]   ttwu_queue_wakelist()
      [p0]   native_smp_send_reschedule()
(3) [p0]   x2apic_send_IPI();
      [p2] smp_reschedule_interrupt()
      [p2] resched_entry: vector=253
      [p2] scheduler_ipi()
(4) [p2]   sched_ttwu_pending()
      [p2]   ttwu_do_activate()
      [p2]   enqueue_task_fair();
      [p2]   ttwu_do_wakeup()
      [p2]   check_preempt_curr()
(5) [p2]   resched_curr();
```



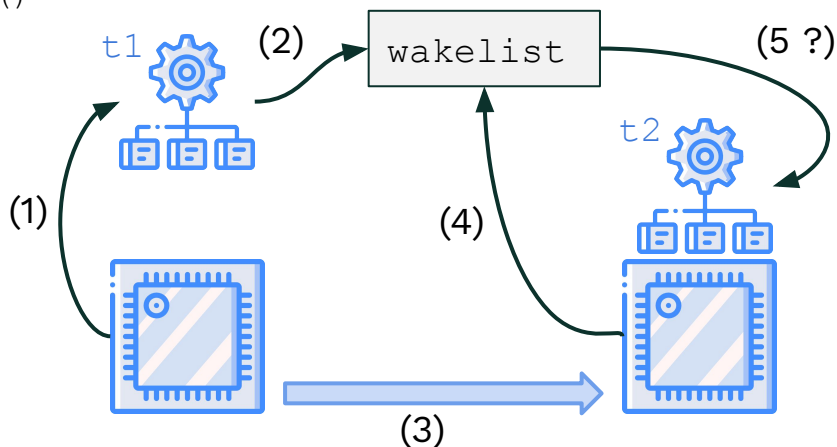
Tracing Wake-ups in VMs

`try_to_wake_up()` in action in a VM:

- p0 (wakeup CPU) and p2 (target CPU), do not share L3 cache (they never do, by default!)
- **p2 is busy**

Pretty much the same!

```
[p0] try_to_wake_up()
(1) [p0] sched_waking: comm=t1 prio=0 target_cpu=p2
(2) [p0]   ttwu_queue_wakelist()
      [p0]   native_smp_send_reschedule()
(3) [p0]   x2apic_send_IPI();
      [p2] smp_reschedule_interrupt()
      [p2] resched_entry: vector=253
      [p2] scheduler_ipi()
(4) [p2]   sched_ttwu_pending()
      [p2]   ttwu_do_activate()
      [p2]   enqueue_task_fair();
      [p2]   ttwu_do_wakeup()
(5) [p2]   check_preempt_curr()
      [p2]   resched_curr();
```

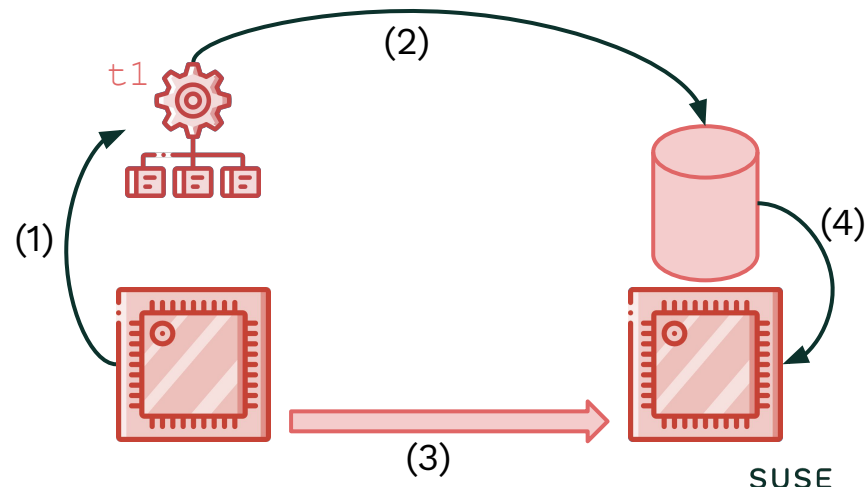


Tracing Wake-ups in VMs

`try_to_wake_up()` in action in a VM:

- p0 (wakeup CPU) and p2 (target CPU), share an L3 cache (defined like that in the Virtual Topology)
- p2 is idle

```
[p0] try_to_wake_up()
(1) [p0] sched_waking: comm=t1 prio=120 target_cpu=p2
[p0]   ttwu_queue_wakelist();
[p0]   ttwu_do_activate()
(2) [p0]   enqueue_task_fair();
[p0]   ttwu_do_wakeup()
[p0]   check_preempt_curr()
[p0]   resched_curr()
[p0]   native_smp_send_reschedule()
(3) [p0]   x2apic_send_IPI();
[p2] smp_reschedule_interrupt()
[p2] reschedule_entry: vector=253
(4) [p2] scheduler_ipi();
```



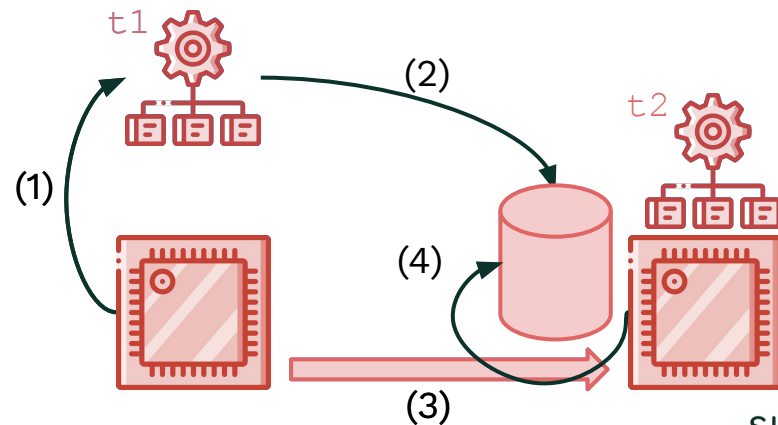
Tracing Wake-ups in VMs

`try_to_wake_up()` in action in a VM:

- p0 (wakeup CPU) and p2 (target CPU), of course (defined like that in the Virtual Topology)
- **p2 is busy**

Pretty much the same!

```
[p0] try_to_wake_up()
(1) [p0] sched_waking: comm=t1 prio= target_cpu=p2
[p0]   ttwu_queue_wakelist();
[p0]   ttwu_do_activate()
(2) [p0]   enqueue_task_fair();
[p0]   ttwu_do_wakeup()
[p0]   check_preempt_curr()
[p0]   resched_curr()
[p0]   native_smp_send_reschedule()
(3) [p0]   x2apic_send_IPI();
[p2] smp_reschedule_interrupt()
[p2] reschedule_entry: vector=253
(4) [p2] scheduler_ipi();
```



Tracing Wake-Ups in VMs

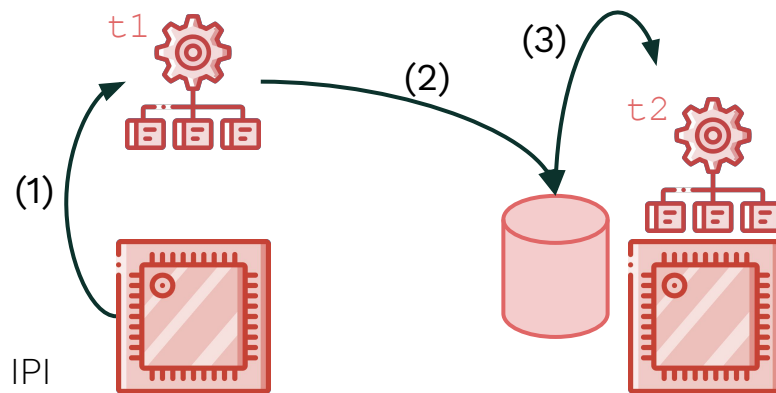
`try_to_wake_up()` in action in a VM:

- p0 (wakeup CPU) and p2 (target CPU), share an L3 cache (defined like that in the Virtual Topology)
- p2 is busy & **t2 has higher priority than t1**

```
[p0] try_to_wake_up()
(1) [p0] sched_waking: comm=t1 prio=120 target_cpu=p2
      [p0]   ttwu_queue_wakelist();
      [p0]   ttwu_do_activate()
(2) [p0]   enqueue_task_fair();
      [p0]   ttwu_do_wakeup()
      [p0]   check_preempt_curr()
      [p0]   check_preempt_wakeup()
      [p0]   wakeup_preempt_entity()
```

⇒ We saved an IPI !!

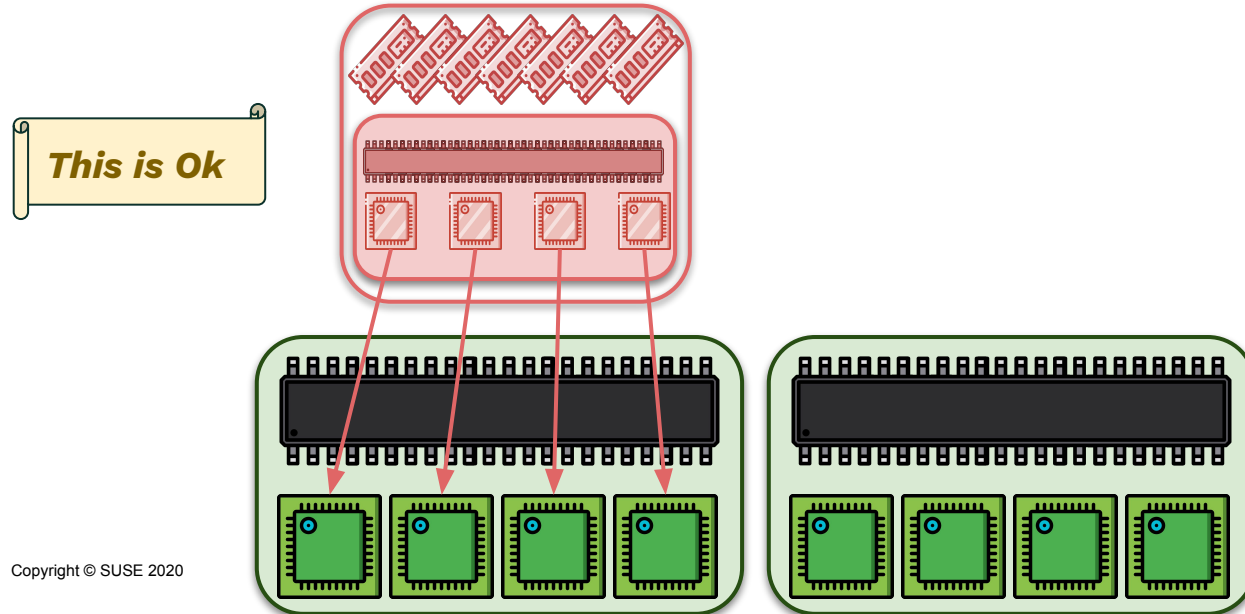
- We avoided disturbing p2 and t2 too much
- In the “no shared L3 configuration” we send the IPI



Is Virtual Good... For Real ?

For real benefits:

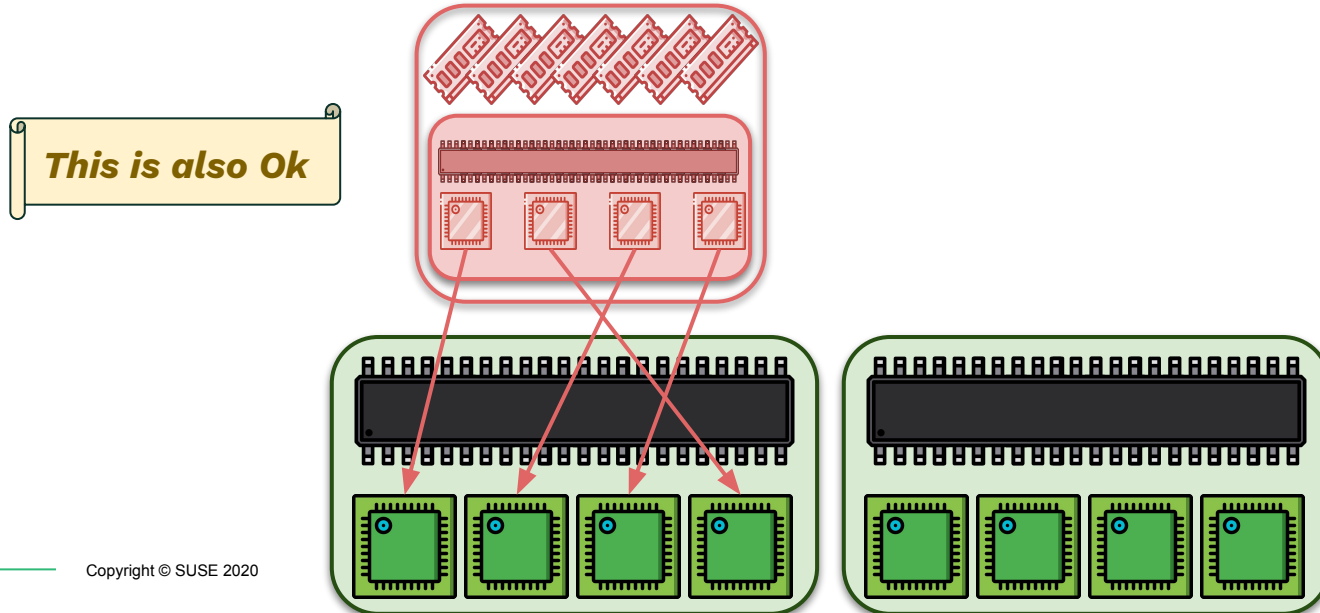
- Virtually shared caches == Really shared caches
- It does not need to be static 1-to-1 resource partitioning



Is Virtual Good... For Real ?

For real benefits:

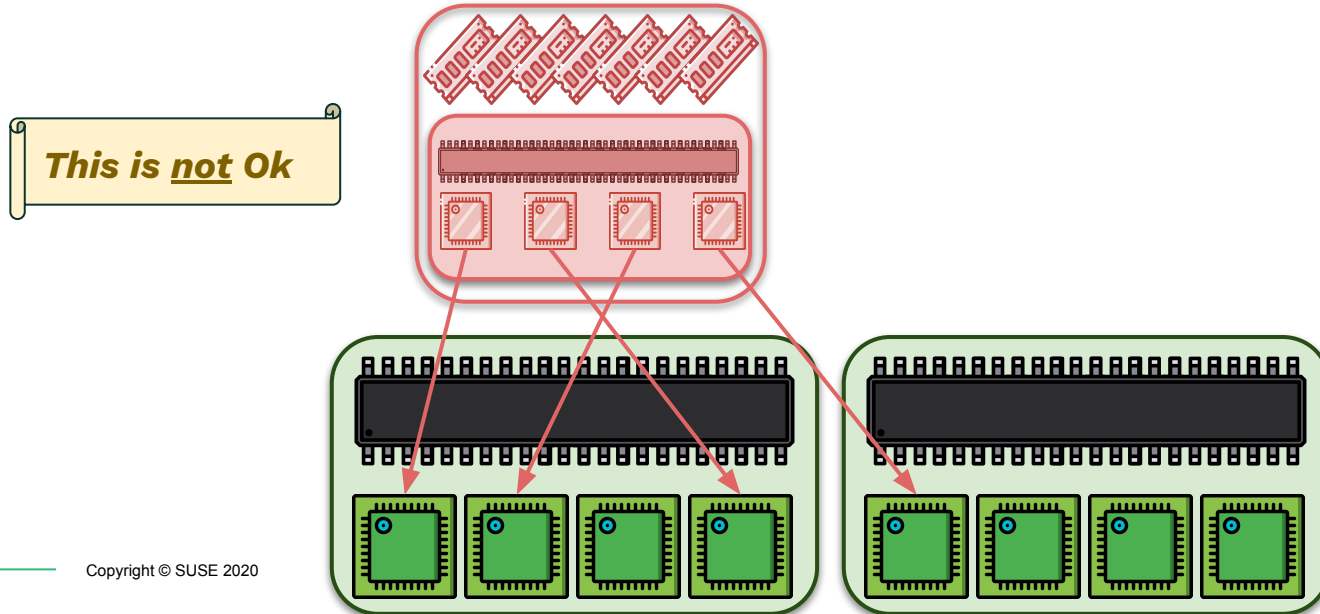
- Virtually shared caches == Really shared caches
- It does not need to be static 1-to-1 resource partitioning



Is Virtual Good... For Real ?

For real benefits:

- Virtually shared caches == Really shared caches
- It does not need to be static 1-to-1 resource partitioning

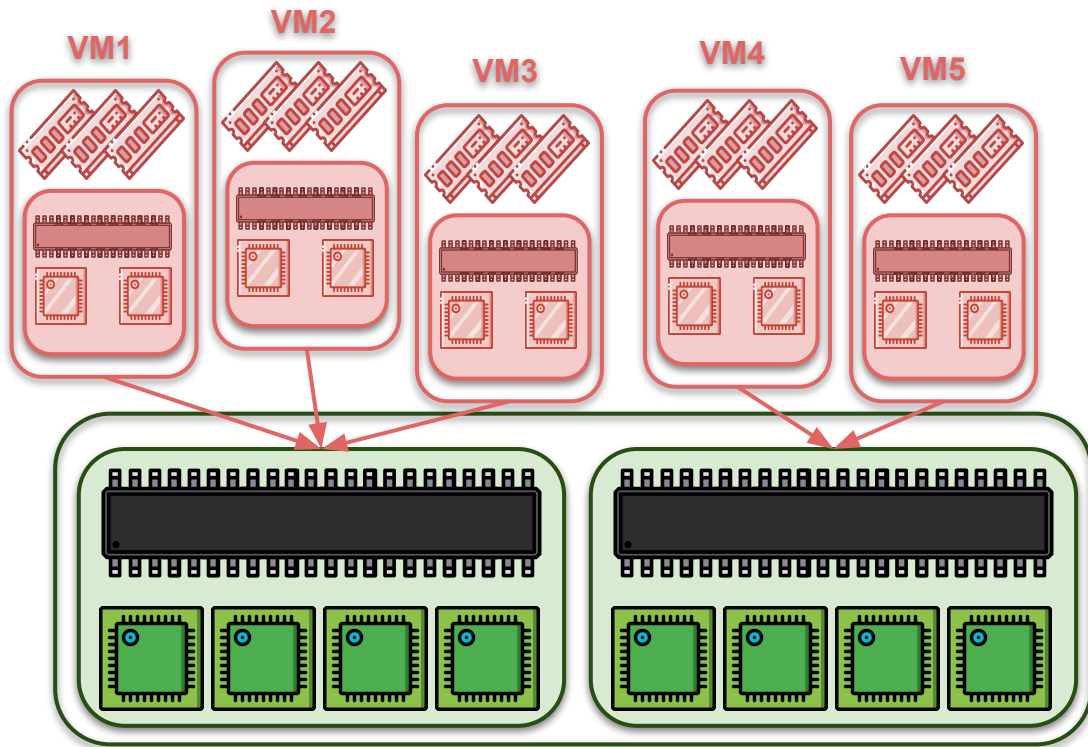


Cache Sharing Consistency

For making sure that vCPUs that share caches run on pCPUs that share caches:

- We don't necessarily need static resource partitioning
- We don't necessarily need 1-to-1 vCPU to pCPU pinning
- We need each VMs to “stick” to some “shared cache domain” on the host (e.g., NUMA nodes)

Theory: with a decently reliable virtual to physical cache sharing relationship is, we will see benefits from cache sharing topologies



“Knock, Knock.”
“Who’s There?”
“It’s Benchmarks!”

Benchmarking Setup

Host:

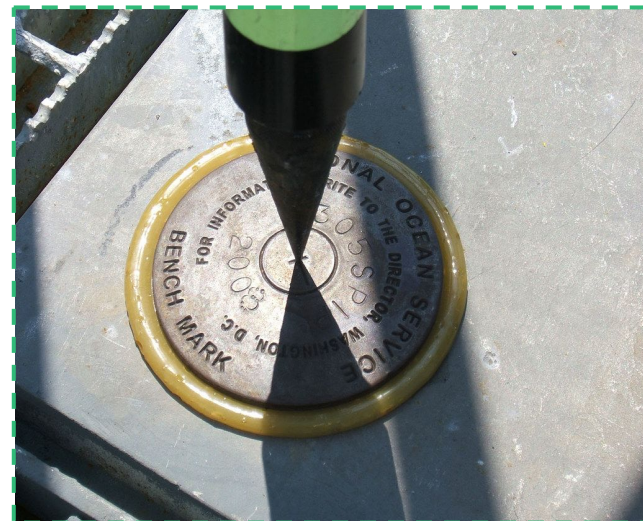
- Xeon(R) Platinum 8260L CPU @ 2.40GHz
- 64 GB RAM, 96 CPUs (2 Sockets, 24 Cores, 2 Threads)
- openSUSE Leap 15.2
- Libvirt and QEMU from latest upstream

VMS:

- 1, 4, 12 and 18 VMs running concurrently
 - 12 VMs run: saturate the host CPUs
 - 18 VMs run: overload
- 4 GB RAM, 8 vCPUs
- openSUSE Leap 15.2

Benchmarking Suite:

- [MMTests](#)



"Sentinel Benchmark" by NOAA's National Ocean Service is licensed under CC BY 2.0

Benchmarks Used

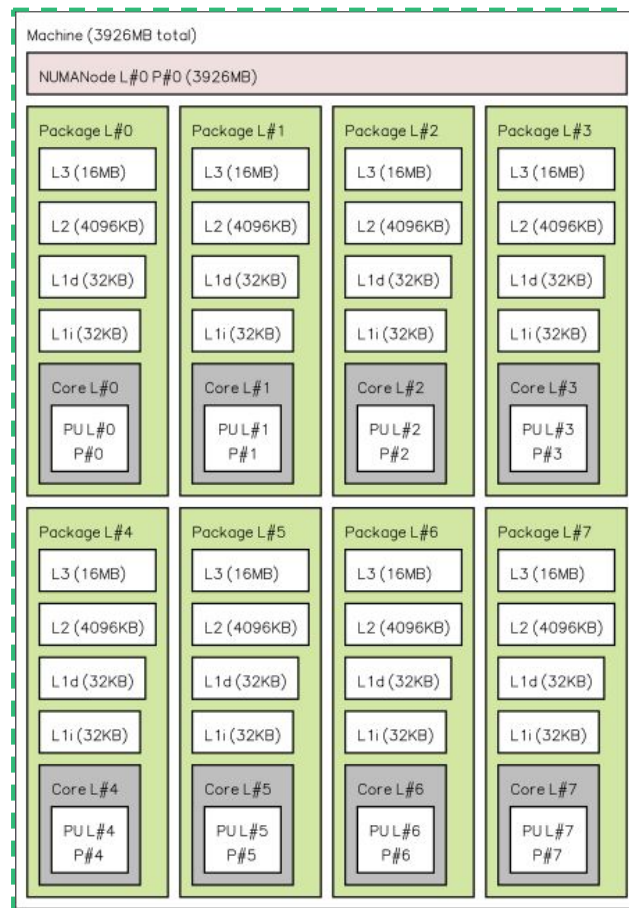
- Hackbench
 - Scheduler benchmark, groups of message passing tasks
 - Threads communicating over pipes
 - Processes communicating over pipes
 - Processes communicating over sockets
 - Number of groups:
 - 1, 3, 5, 7, 12, 18, 24, 30, 32
 - Measures the wake-up latency
- Perfpipes
 - Basically `perf bench sched pipe` (based on Ingo Molnar [pipe-test-1m.c](#))
 - 2 processes, ping-pong messages over a pipe for thousands loops
 - Measures total durations
- Schbench
 - Message passing between clients / server:
 - 1 server
 - 1, 2, 4, 7 clients
 - Measures 99.0-th percentile of wake-up latency

Benchmarked Configuration

- default
 - No pinning at all
 - All vCPUs of all VMs are free to move around all pCPUs
- Numad
 - “Automatic” vCPU and memory placement, done via numad
 - No explicit pinning, but numad should limit cross-NUMA node movement
- perNODE
 - Each VM were pinned to a node
 - All vCPUs of the VM were pinned to all pCPUs of the node (no 1-to-1)
 - Odd VMs, node 0. Even VMs, node 1
- 1-to-1
 - Classic static resource partitioning approach
 - Each vCPU was pinned to 1 pCPU
 - No two vCPUs pinned to the same pCPU
 - (skipped the run with 18 VMs)

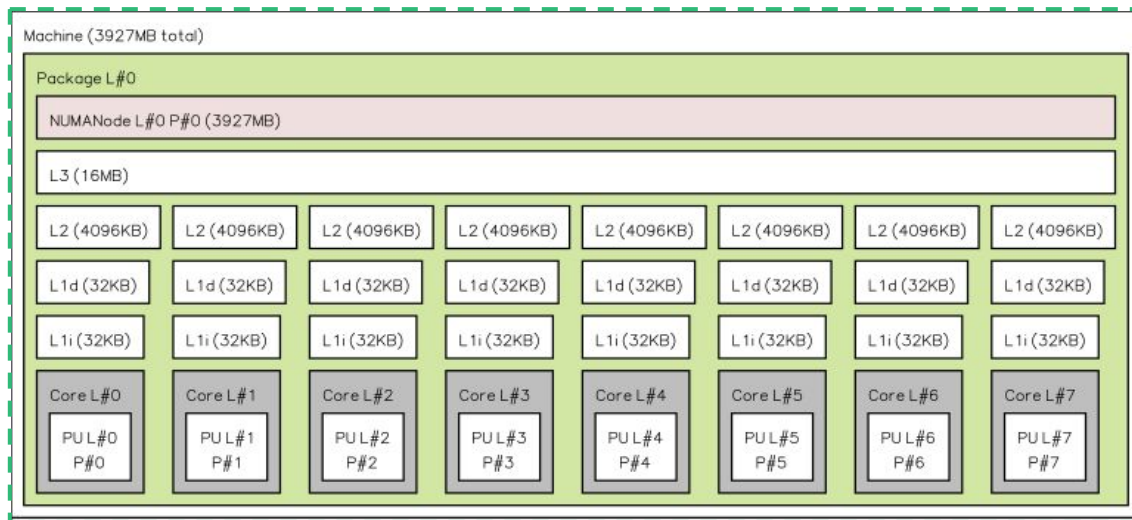
Benchmarked Topologies

- 8 sockets
 - Default topology
 - Each vCPU was a socket
 - No cache sharing between any vCPU



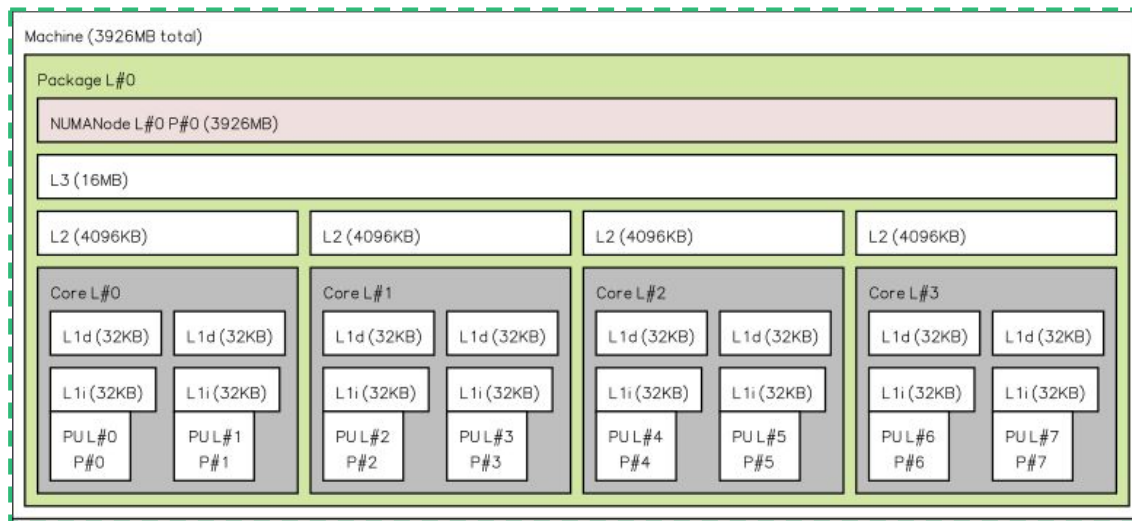
Benchmarked Topologies

- 8 cores - 1 thread
 - All vCPUs are cores of one unique socket
 - They all share an L3 cache
 - No (virtual) hyperthreading




Benchmarked Topologies

- 4 cores - 2 threads
 - All vCPUs are part of one unique socket
 - They are arranged in 4 cores
 - Each core has 2 threads



Benchmarked Topologies

- 8 sockets
 - Default topology
 - Each vCPU was a socket
 - No cache sharing between any vCPU
- 8 cores - 1 thread
 - All vCPUs are cores of one unique socket
 - They all share an L3 cache
 - No (virtual) hyperthreading
- 4 cores - 2 threads
 - All vCPUs are part of one unique socket
 - They are arranged in 4 cores
 - Each core has 2 threads



When all the vCPUs run on the same node, Virtual and Real cache sharing matches

Benchmarked Topologies

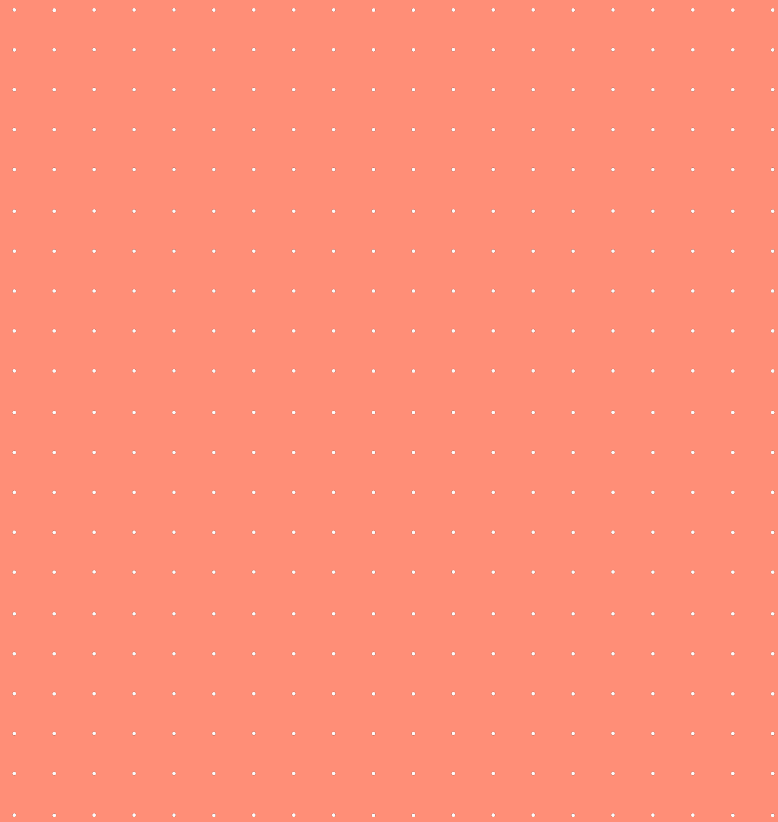
- 8 sockets
 - Default topology
 - Each vCPU was a socket
 - No cache sharing between any vCPU
- 8 cores - 1 thread
 - All vCPUs are cores of one unique socket
 - They all share an L3 cache
 - No (virtual) hyperthreading

- 4 cores - 2 threads
 - All vCPUs are part of one unique socket
 - They are arranged in 4 cores
 - Each core has 2 threads



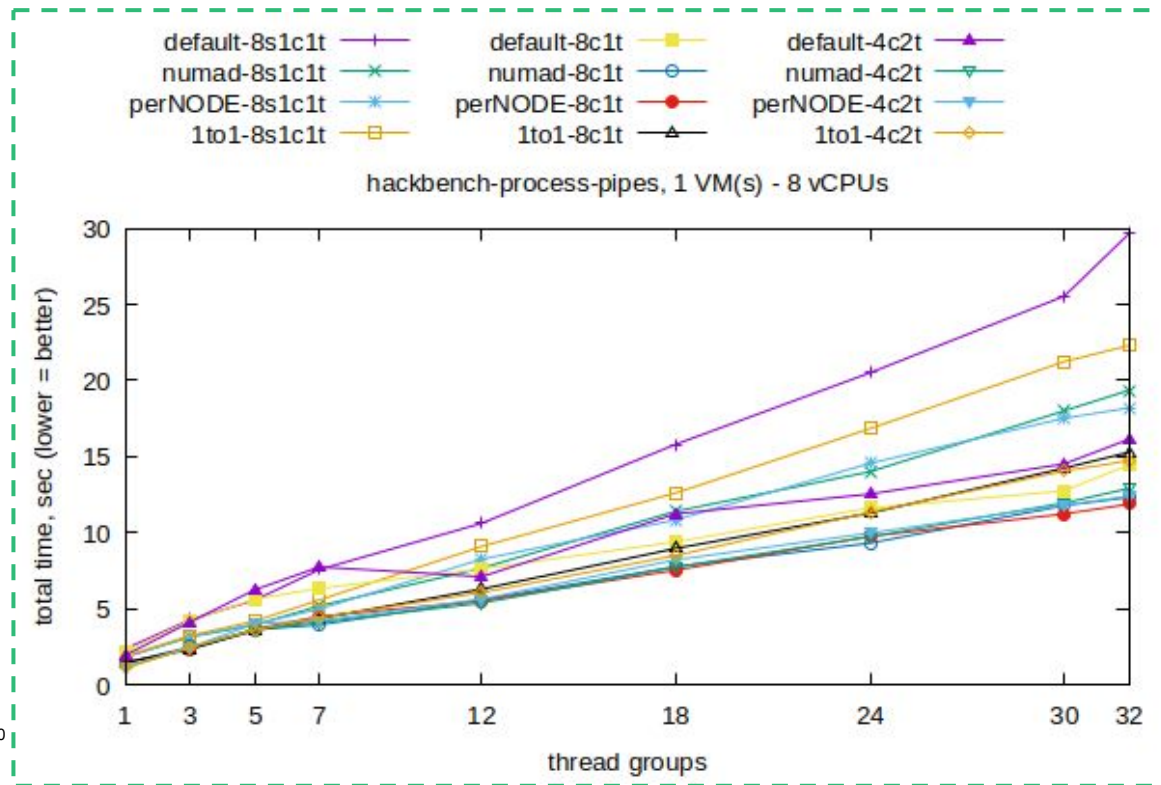
When doing 1-to-1 vCPU pinning, we achieve static & dedicated resource partitioning with matching topology

Hackbench



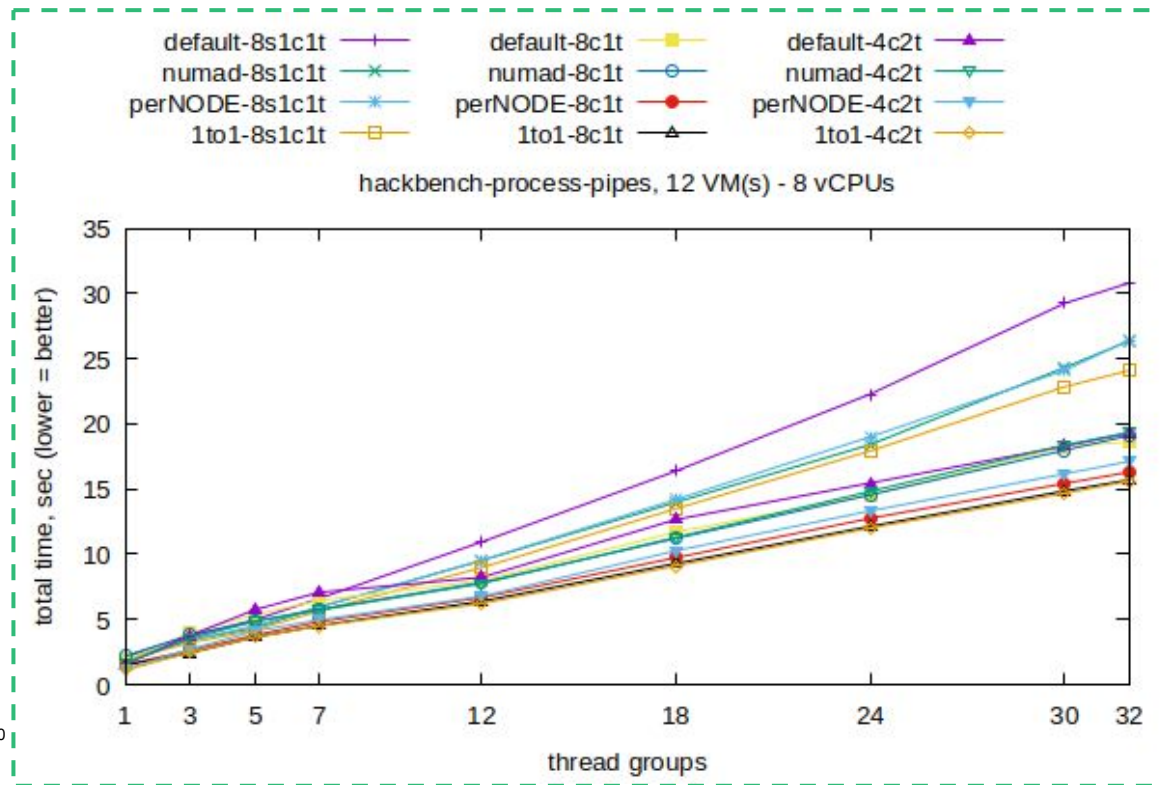
Benchmarks Results

- Hackbench



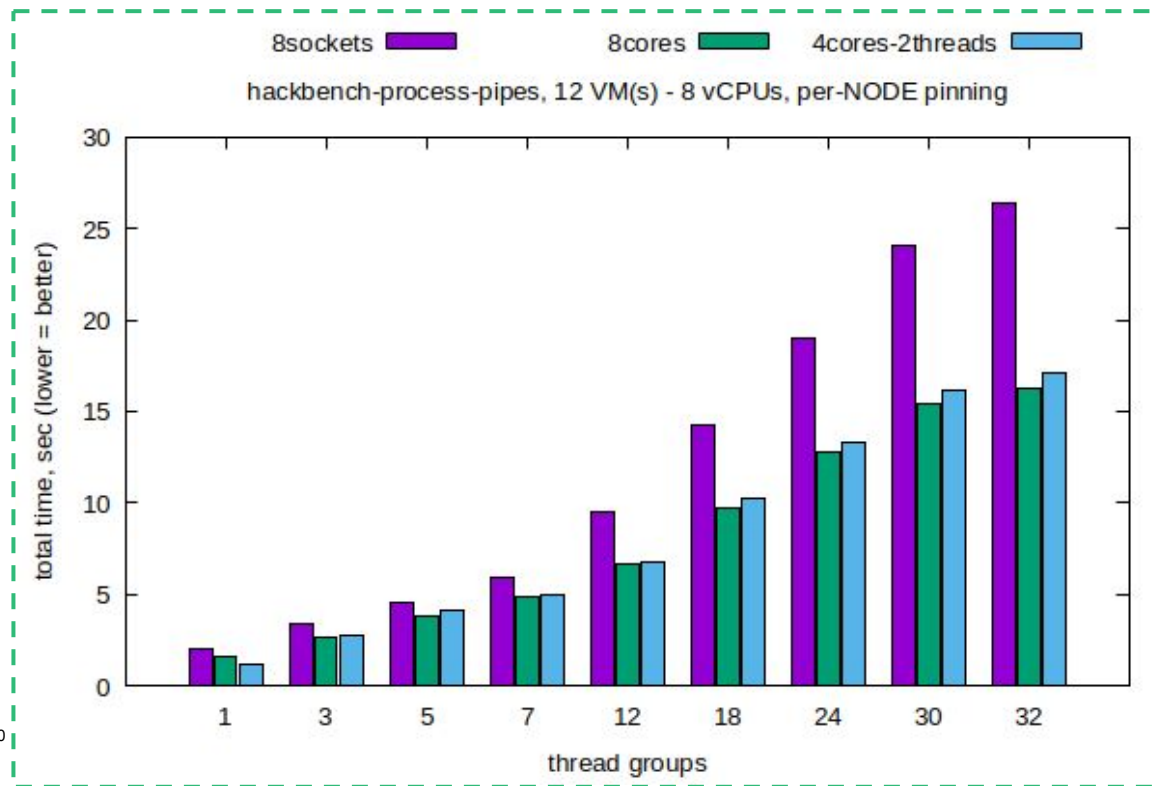
Benchmarks Results

- Hackbench



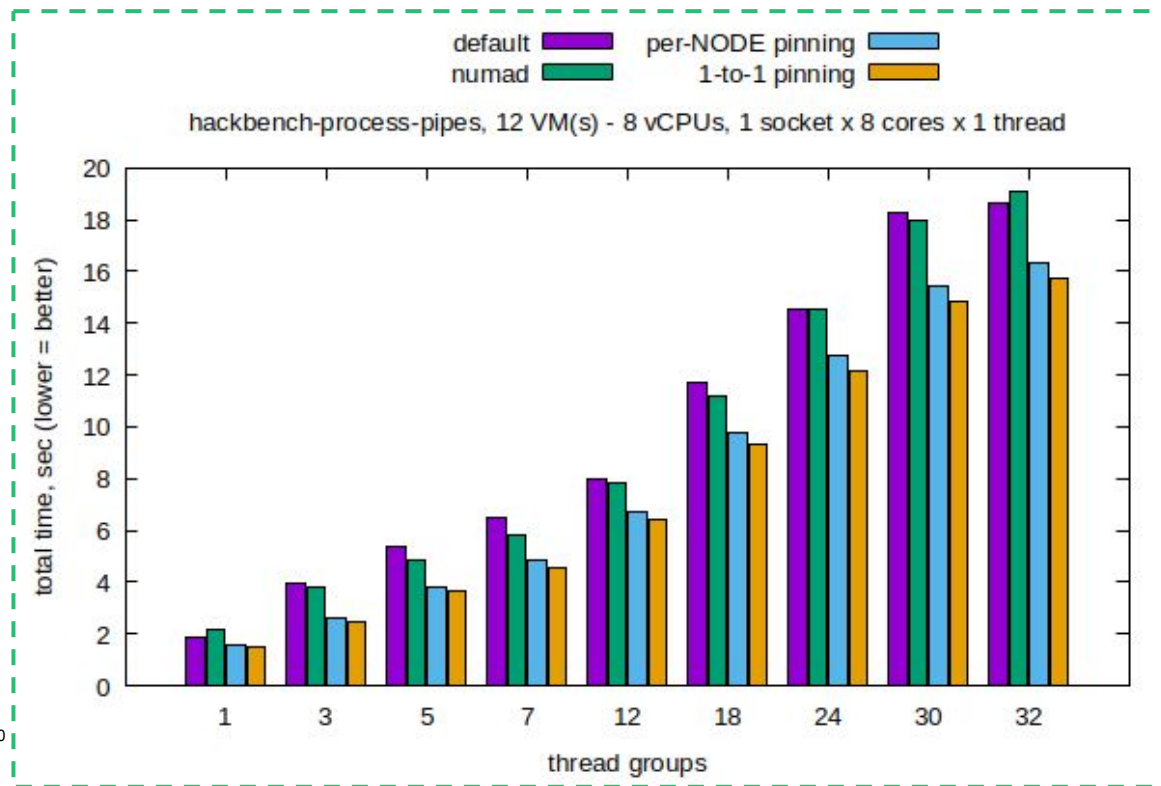
Benchmarks Results

- Hackbench



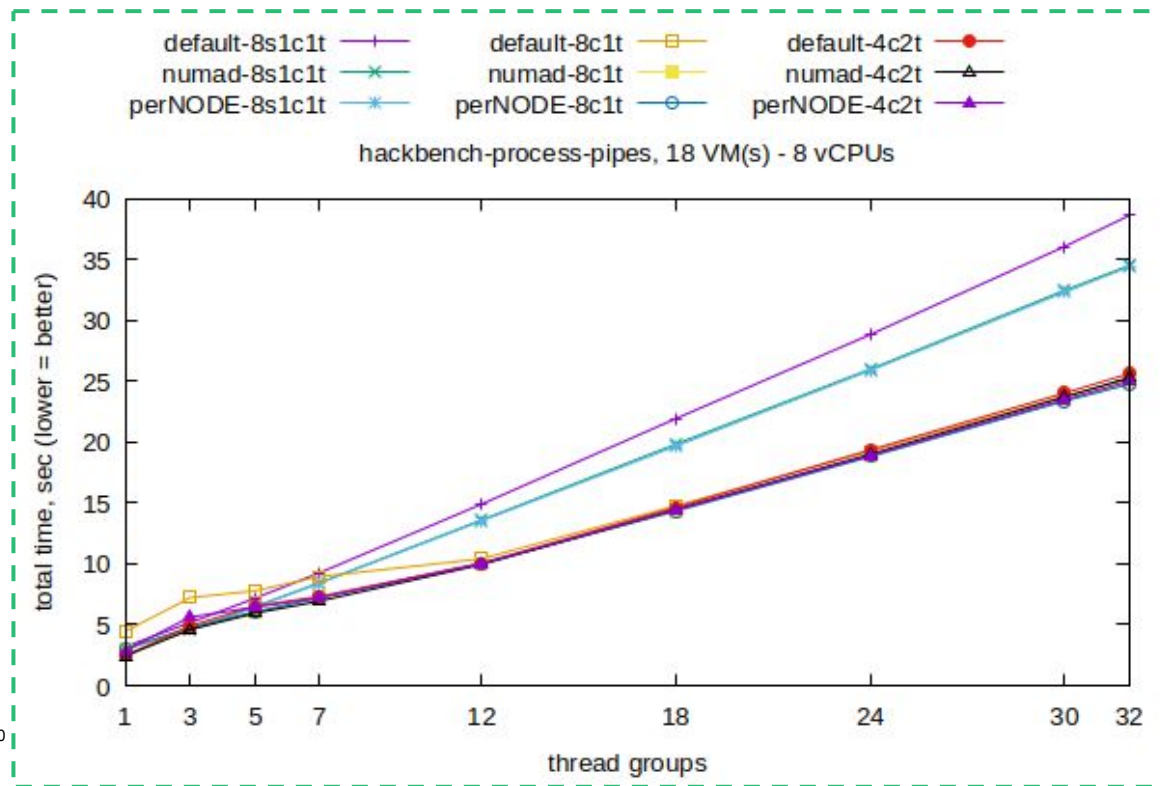
Benchmarks Results

- Hackbench

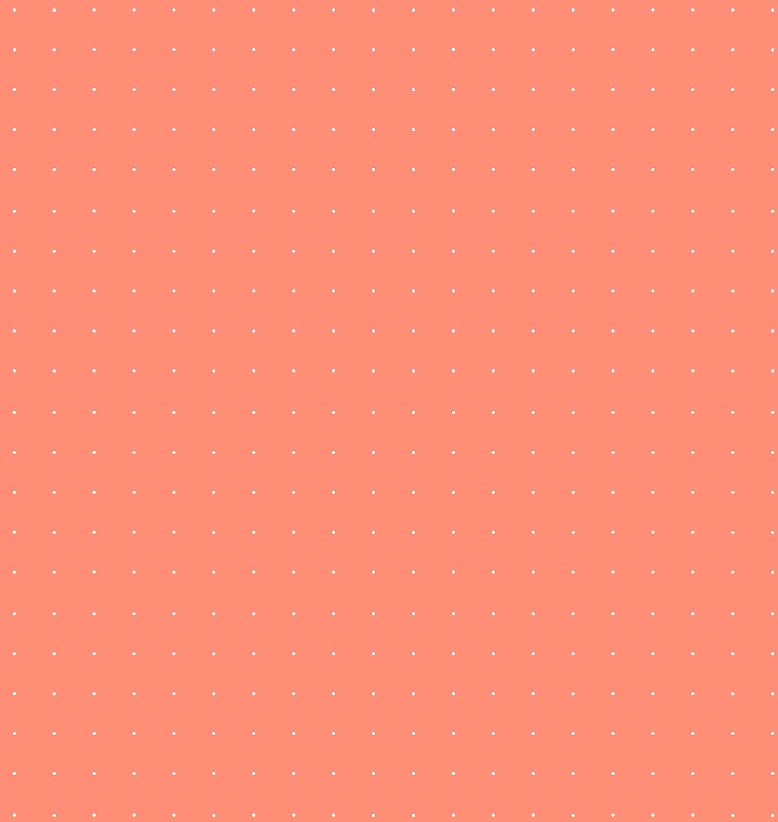


Benchmarks Results

- Hackbench

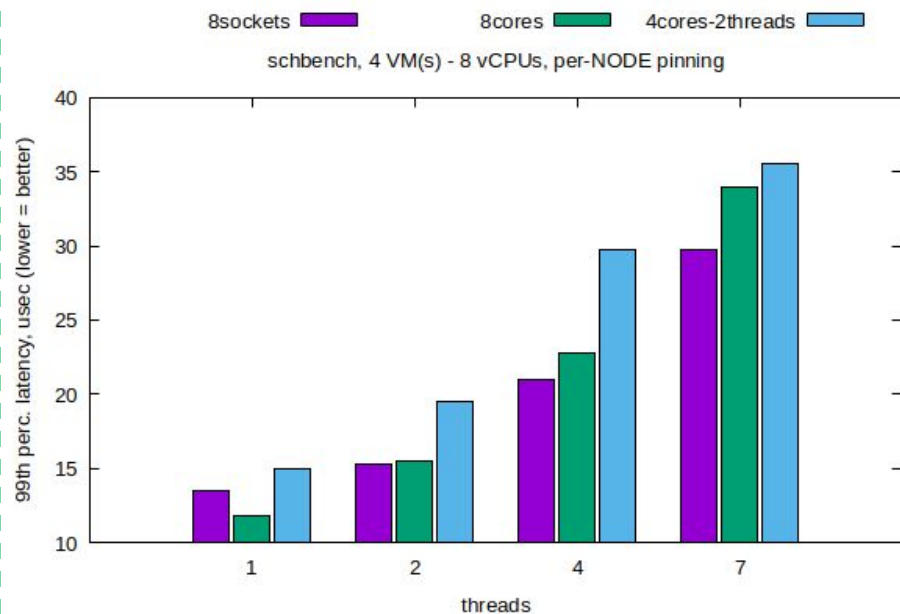
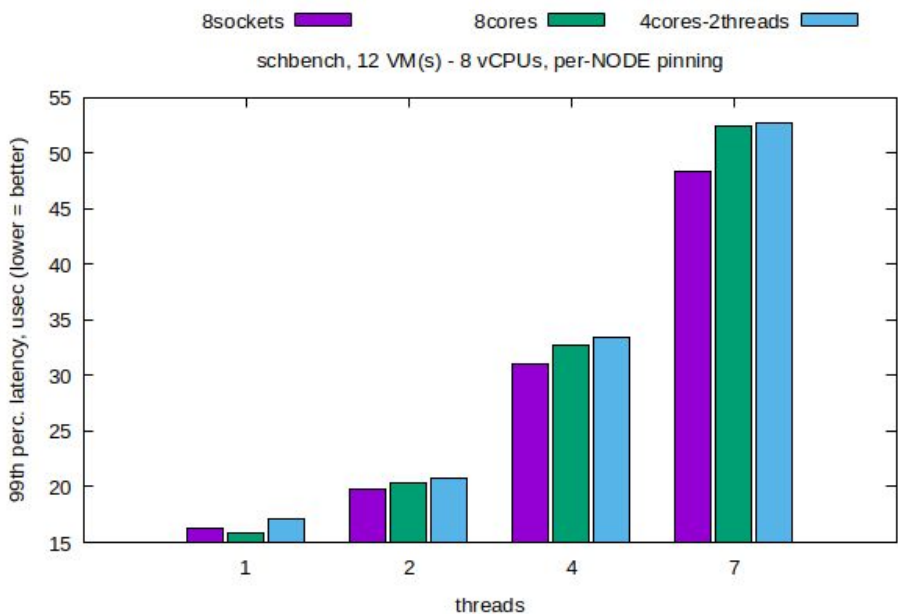


Schbench



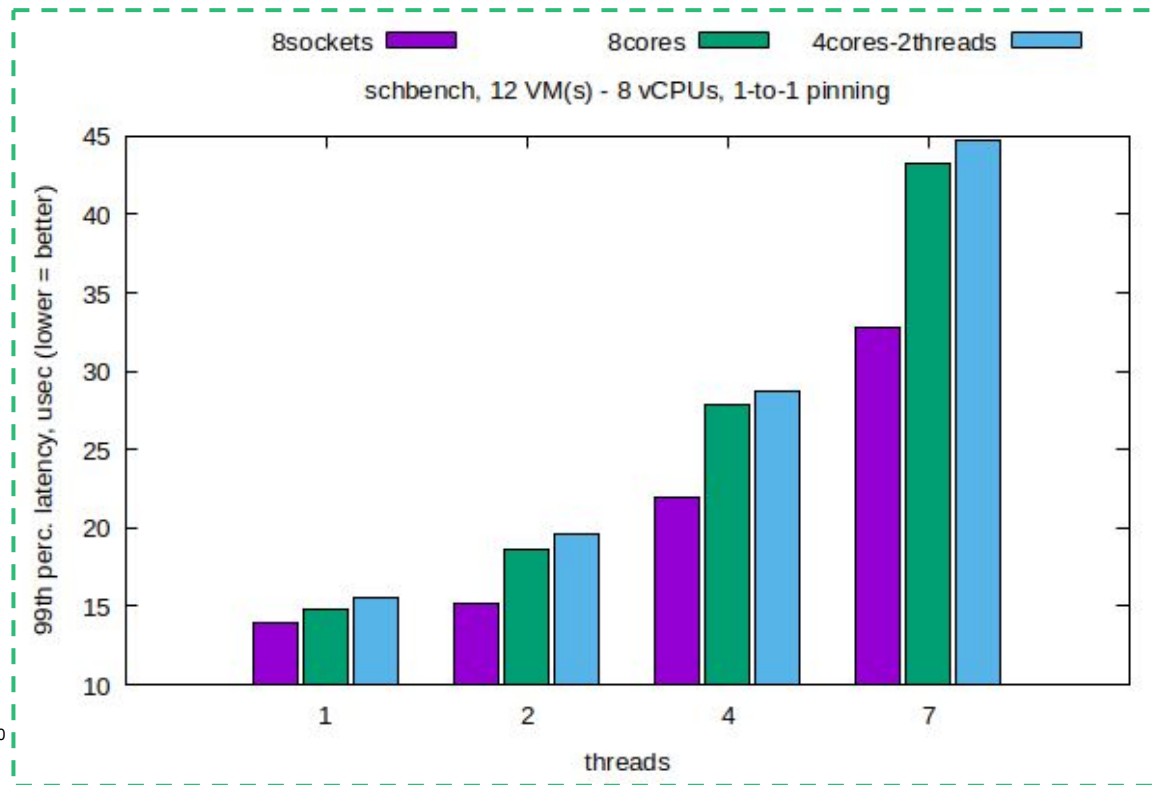
Benchmarks Results

- Schbench

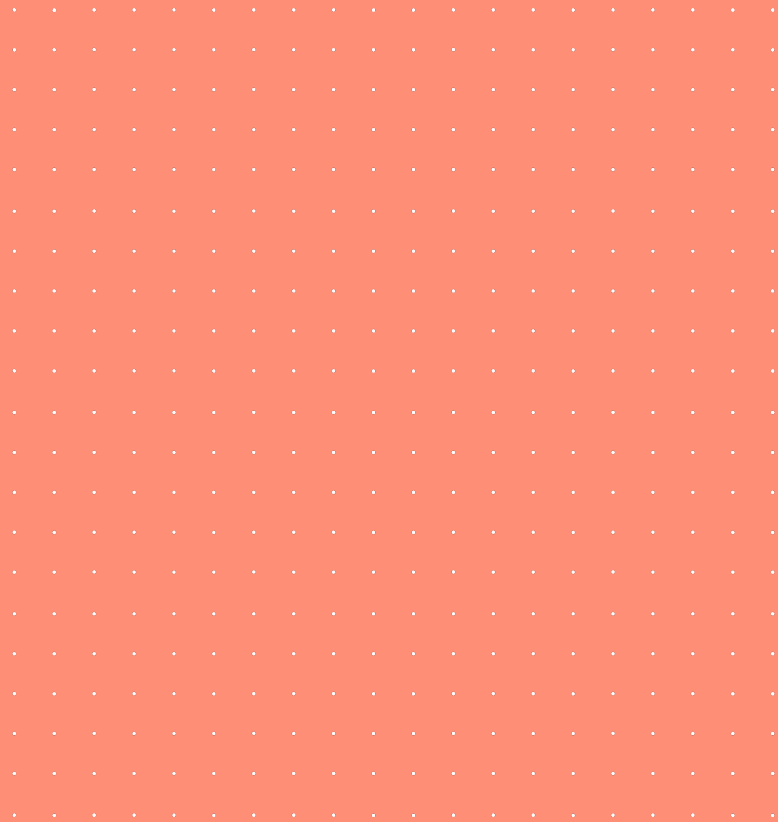


Benchmarks Results

- Schbench



Perfpipe

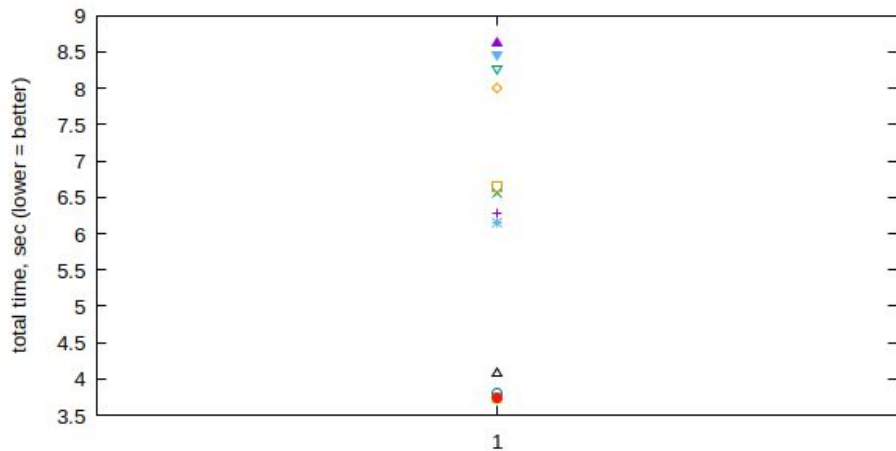


Benchmarks Results

- Perfpipes

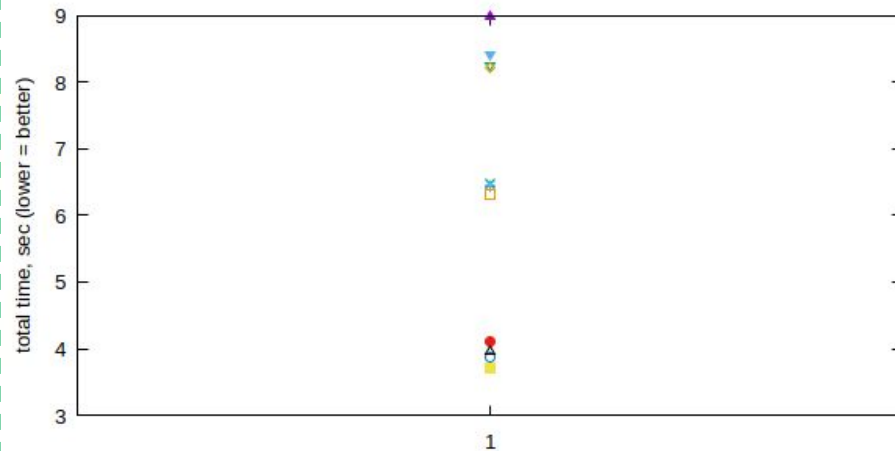
default-8s1c1t default-8c1t default-4c2t
 numad-8s1c1t numad-8c1t numad-4c2t
 perNODE-8s1c1t perNODE-8c1t perNODE-4c2t
 1to1-8s1c1t 1to1-8c1t 1to1-4c2t

perfpipes, 1 VM(s) - 8 vCPUs



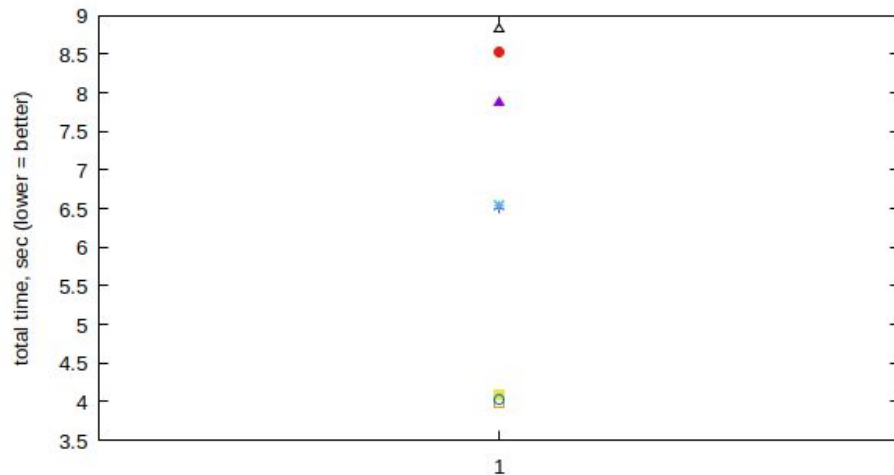
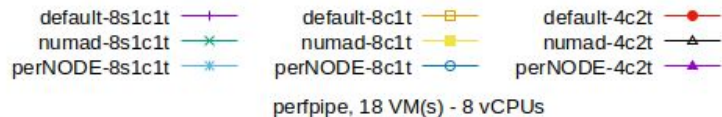
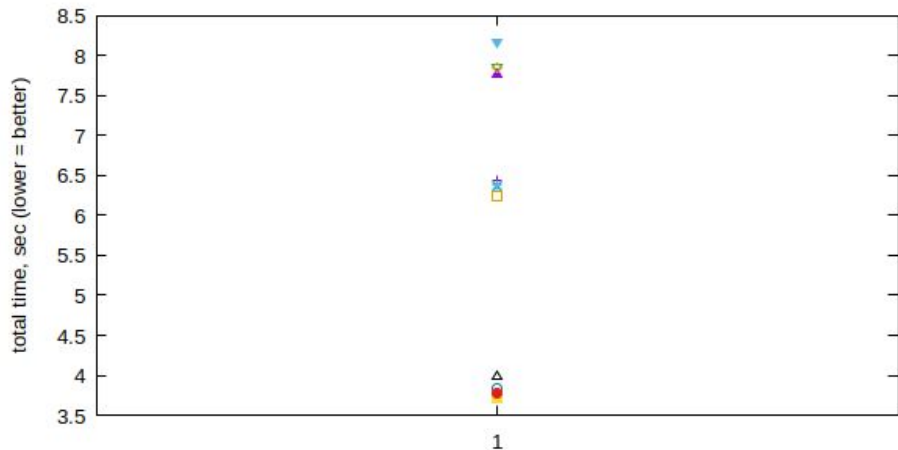
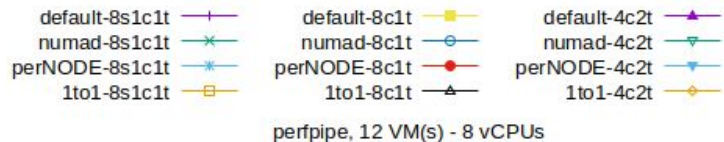
default-8s1c1t default-8c1t default-4c2t
 numad-8s1c1t numad-8c1t numad-4c2t
 perNODE-8s1c1t perNODE-8c1t perNODE-4c2t
 1to1-8s1c1t 1to1-8c1t 1to1-4c2t

perfpipes, 4 VM(s) - 8 vCPUs

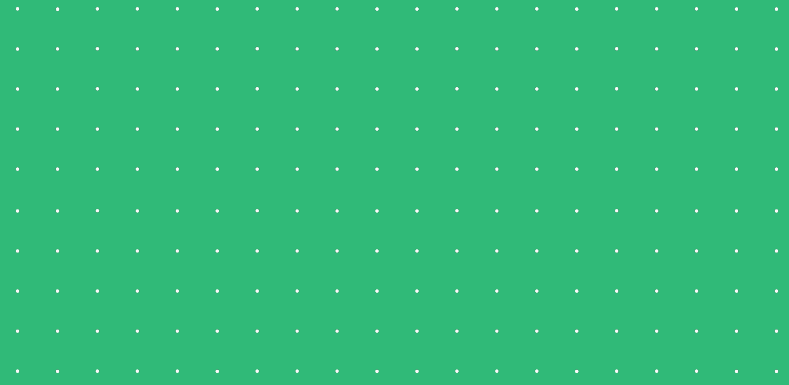


Benchmarks Results

- Perfpip



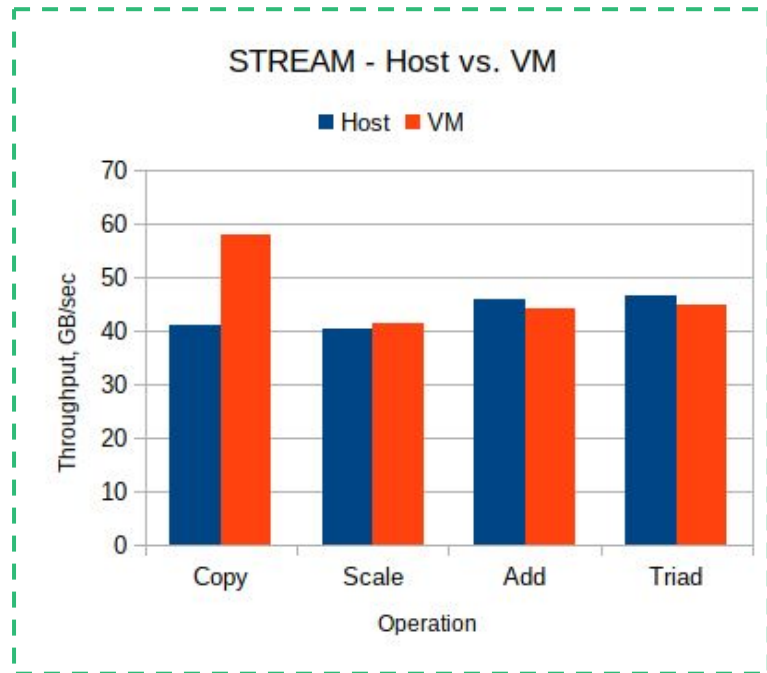
Disturbing Graphs (More of Them!)



STREAM Quality Degraded

After tuning, we expected comparable [STREAM](#) performance in Host and VM

- Accomplished for Scale, Add, Triadd operations
- Why Copy is different?
- Why the VM is faster?



STREAM Quality Degraded

After tuning, we expected comparable STREAM performance in Host and VM

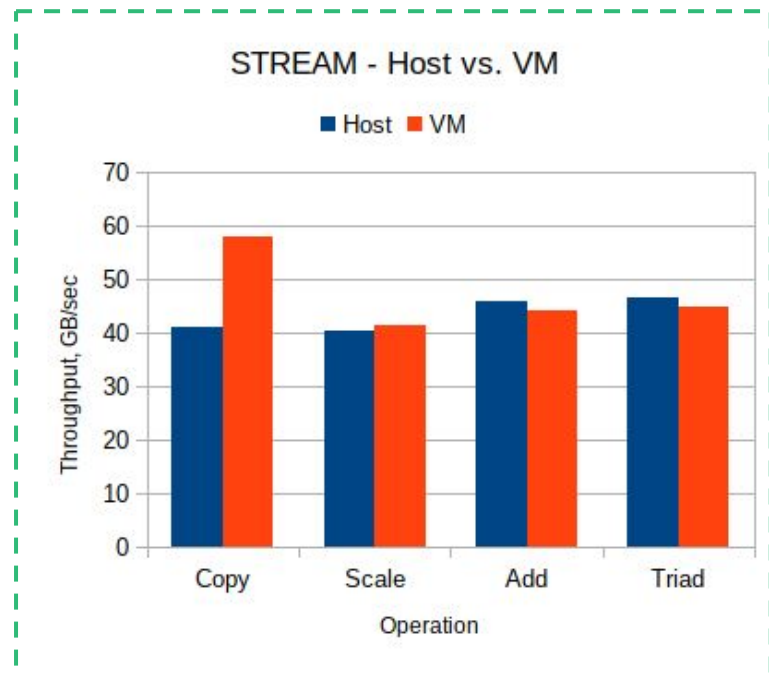
- Accomplished for Scale, Add, Triadd operations
- Why Copy is different?
- Why the VM is faster?

After some serious head-scratching, we found this:

```
perf stat -e r44B ./stream
```

⇒ code for `PREFETCHNTA` instr. being used

- Host: 0 events
- VM: 125213413 events



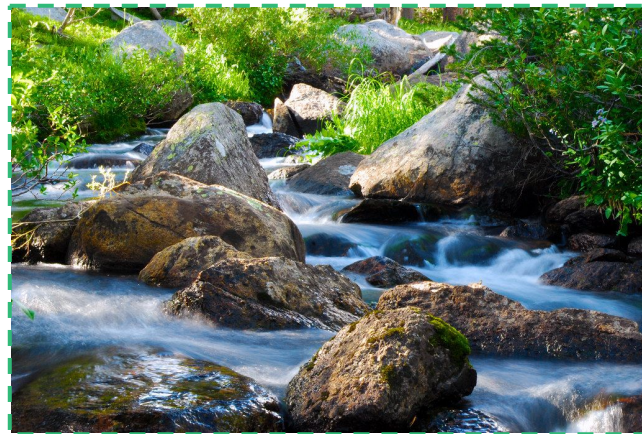
STREAM Copy Op Dissected

STREAM Copy Op:

- `memcpy()` across 3 arrays, 763 MB each
- Total memory 2289 MB
- 128 threads (configurable)

Glibc `memcpy()` implementation [*]

- $(L3_cache_size / nr_cores + L2_cache_size) * 4 = K$
 - If copying more than K bytes
 - \Rightarrow use `PREFETCHNTA`
 - If copying less than K bytes
 - \Rightarrow **do not** use `PREFETCHNTA`



"Wind River Stream" by Kylir is licensed under CC BY 2.0

(Virtual) Topology role starting to become clear...

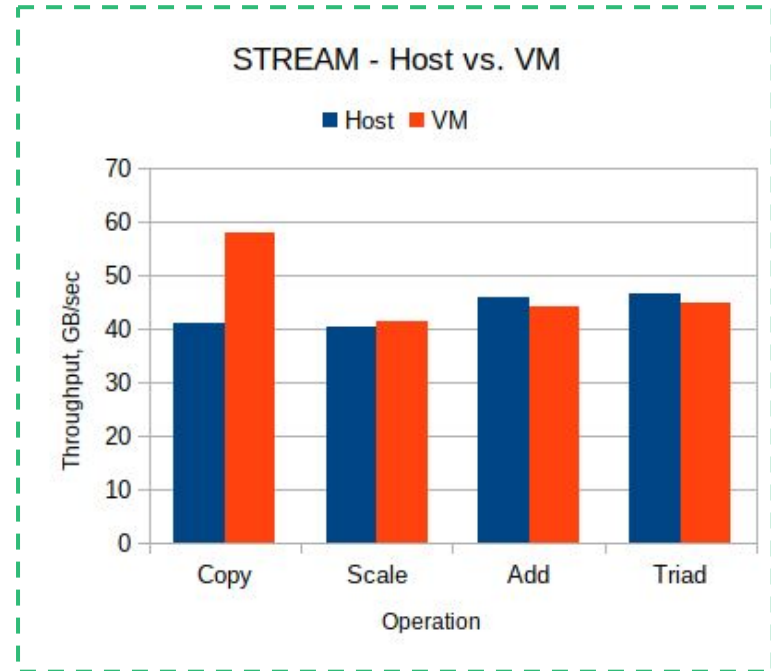
Virtual Topology & Cache (Again!)

Baremetal:

- 64 cores, L3 cache is 64 MB
 - $(64 \text{ MB} / 64 + 512 \text{ KB}) * 4 = 1.5 \text{ MB} * 4 = \mathbf{6 \text{ MB}}$
- Each memcopy()
 - Array size / nr_threads
 - $763 \text{ MB} / 128 \approx 5.96 \text{ MB} < 6 \text{ MB}$
- \Rightarrow does not use PREFETCHNTA

VM:

- 64 cores, L3 cache is 16 MB (QEMU's default)
 - $(16 \text{ MB} / 64 + 512\text{K}) * 4 = 0.75 \text{ MB} * 4 = \mathbf{3 \text{ MB}}$
 - $763 \text{ MB} / 128 \approx 5.96 \text{ MB} > 3 \text{ MB}$
- \Rightarrow uses PREFETCHNTA



Too Big or Too Small Caches

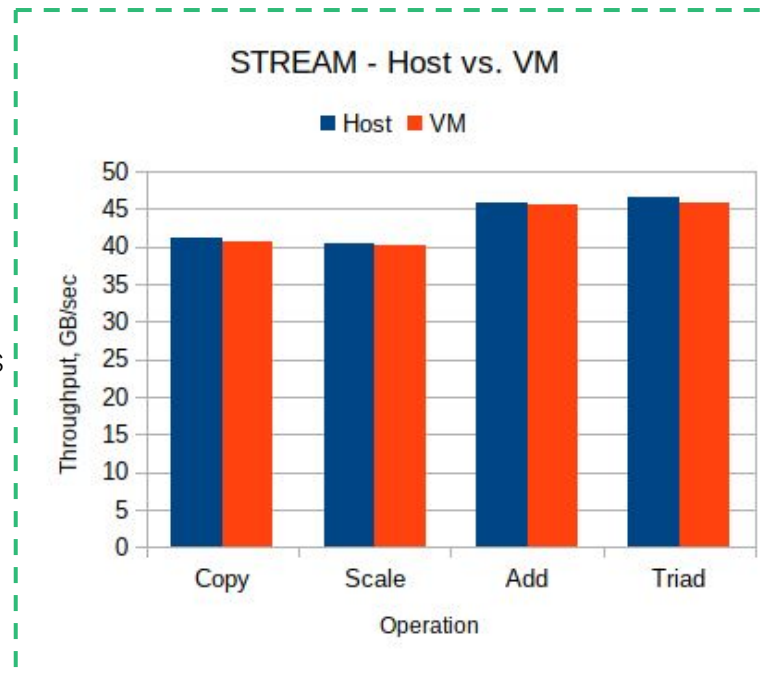
In this case, “fix” is easy

- `<cache mode='passthrough' />`
- “Let’s make the VM slow again!” ;-P

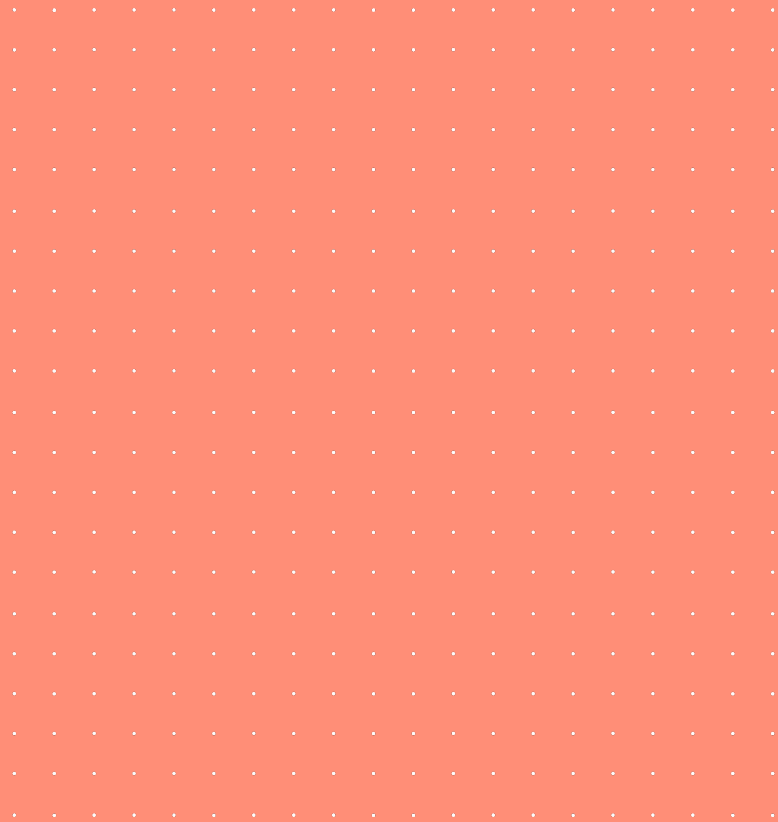
But in general:

- With `'mode=emulate'`, we may have VMs with lots of vCPUs, but really small caches
 - E.g., (as in this case): 128 vCPUs, 16 MB L3
- With `'mode=passthrough'`, we may have small VMs with just a few vCPUS, and gigantic caches
 - E.g., a 2 vCPUs VM on this host: 2 vCPUs, 64 MB L3

⇒ Does this calls for adding mechanisms for specifying the cache hierarchy in details in the Virtual Topology?



Topology That Makes you Vulnerable?



How Many ~~Engineers~~ Bits Does It Take To...

Once upon a time, in a ~2.7 TB VM

- Check mitigations:

```
Itlb_multihit: Not affected
l1tf: Vulnerable
[...]
```

- Check dmesg:

```
L1TF: System has more than MAX_PA/2 memory.
      L1TF mitigation not effective
```

- Check lscpu

```
Address sizes: 42 bits physical, 48 bits virtual
```

- Check dmesg again

```
BIOS-e820: [mem 0x0000000100000000-0x000002b57fffffff] usable
```



"Changing Lightbulbs" by shareski is licensed under CC BY-NC 2.0

How Many ~~Engineers~~ Bits Does It Take To...

The Problem:

- e820 ends at `0x2b57ffff` ⇒ needs more than 41 bits
 - `MAX_PA` is on 42 bit
 - `MAX_PA/2` (for PTE inversion, for L1TF) is 41 bits
- Too much memory for too few bits (host has 46)
 - The kernel informs us that we may be vulnerable
 - (We're fine! But still, it's annoying)

Solution:

- Give the guest more bits
 - Same as the host (as we're using `host-passthrough` CPU model)

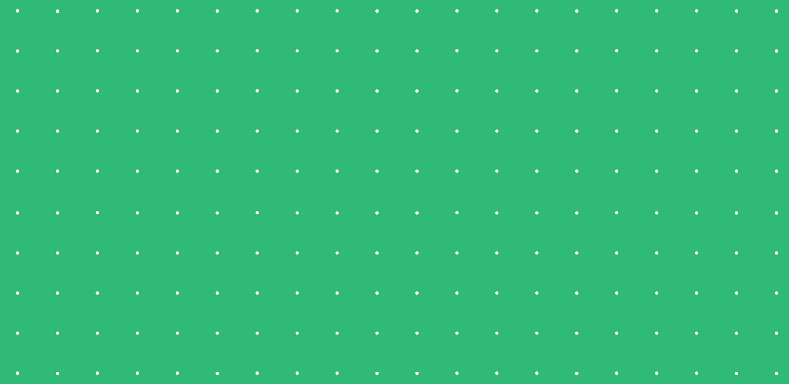
```
</devices>
  <qemu:commandline>
    <qemu:arg value='-cpu' />
    <qemu:arg value='host,host-phys-bits=on' />
  </qemu:commandline>
```

Hack-ish... Working on patches



"Changing Lightbulbs" by shareski is licensed under CC BY-NC 2.0

Conclusions



Conclusions

- Topology, Physical and Virtual, has probably more of an impact that what you would expect
- Software algorithms, thresholds and optimizations are done thinking to real existing hardware. Let's keep this in mind and work on Virtual Topology, or it will bite us (again!)
- Maybe there are benefits in using Virtual Topology in VMs, even when we don't have statical and dedicated resource partitioning



"Handshakes" by Ron Cogswell is licensed under CC BY 2.0

Dario Faggioli

- Ph.D on Real-Time Scheduling, soft real-time scheduling in Linux SCHED_DEADLINE
- 2011, Sr. Software Engineer @ Citrix The Xen-Project, hypervisor internals, NUMA-aware scheduler, Credit2 scheduler, Xen scheduler maintainer
- 2018, Virtualization Software Engineer @ SUSE Xen, KVM, QEMU, Libvirt; core-scheduling, performance evaluation & tuning
- Mail: <dfaggioli@suse.com>
Twitter: [@DarioFaggioli](https://twitter.com/DarioFaggioli)
IRC: dariof



© 2020 SUSE LLC. All Rights Reserved.
SUSE and the SUSE logo are registered trademarks of
SUSE LLC in the United States and other countries.
All third-party trademarks are the property of their
respective owners.

For more information, contact SUSE at:
+1 800 796 3700 (U.S./Canada)
+49 (0)911-740 53-0 (Worldwide)

SUSE
Maxfeldstrasse
90409 Nuremberg
www.suse.com

Thank you!

Questions?

