Micro-Optimizing KVM VM-Exits

KVM Forum 2019
Nov 1 2019 – Lyon, France

Andrea Arcangeli <aarcange at redhat.com>
Distinguished Engineer

Agenda

- Example of problematic workloads to virtualize efficiently that currently trigger frequent VM-Exits with upstream KVM
- Recap of the different kinds of speculative execution attacks and mitigations
  - Not about HT/SMT (orthogonal and not enough time)
- Benchmarks of the current (kernel v5.3) speculative execution mitigations on VMX and SVM
- Two proposals to micro optimize the KVM VM-Exits in the host

Hard to virtualize guest workloads

- The most effective way optimize the guest mode is to reduce the number of VM-Exits with:
  - device-assignment for I/O with hardware devices (VFIO, IOMMU, VT-d, SR-IOV, ...)
    - ✓ Network | storage (NVME/SSD/SCSI) | GPU | RDMA
  - vhost-user-blk/scsi/net for virtualized I/O
- Some guest workloads will still flood KVM with VM-Exits, for example:
  - **Guest scheduling events on idle vCPUs**
    - ✓ *cpuidle-haltpoll* upstream guest idle governor makes this case a lesser concern
      - ✗ It *risks wasting CPU* in guest mode if the host isn't idle
  - **Guest high resolution timers**

Guest scheduling events on idle vCPUs

```
if (fork()) {
        while (n--) {
                read(pipe1[0], buf, 1);
                write(pipe2[1], buf, 1);
        }
        wait(NULL);
} else {
        while (n--) {
                write(pipe1[1], buf, 1);
                read(pipe2[0], buf, 1);
        }
}
```

```
perf kvm stat record -a sleep 1

            HOT CAN_GET_HOT ????

             VM-EXIT    Samples  Samples%
           MSR_WRITE     605044    75.08%
                 HLT     199774    24.79%
  EXTERNAL_INTERRUPT        494     0.06%
    PREEMPTION_TIMER        297     0.04%
   PENDING_INTERRUPT        290     0.04%
            MSR_READ          8     0.00%
       EPT_MISCONFIG          6     0.00%
   PAUSE_INSTRUCTION          3     0.00%
```

Guest high resolution timers

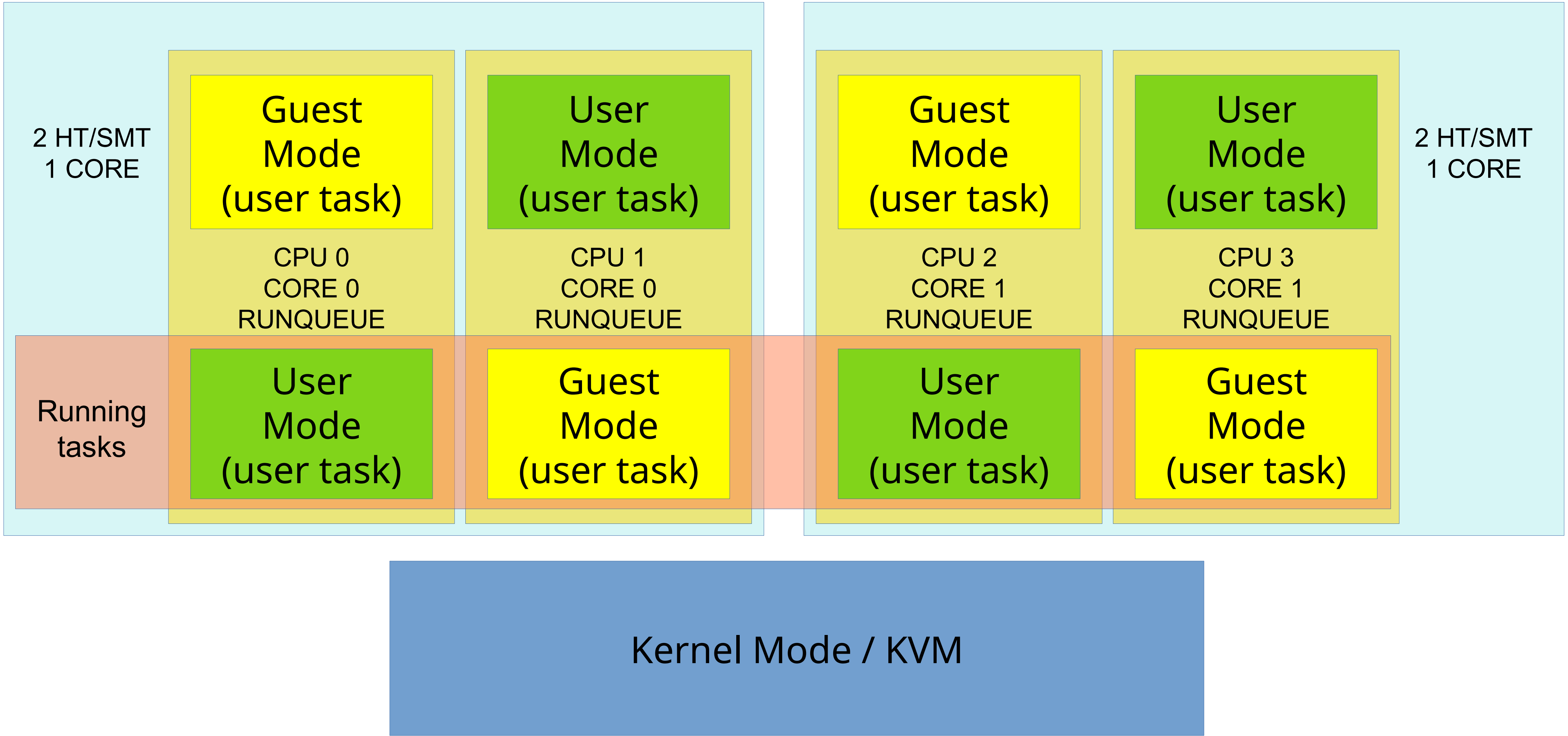
```
sigevent.sigev_notify = SIGEV_SIGNAL;
sigevent.sigev_signo = SIGALRM;
sigevent.sigev_value.sival_ptr = &timer;
if (timer_create(CLOCK_REALTIME, &sigevent, &timer) < 0)
        perror("timer_create"), exit(1);

itimerspec.it_value.tv_sec = 0;
itimerspec.it_value.tv_nsec = 1;
itimerspec.it_interval.tv_sec = 0;
itimerspec.it_interval.tv_nsec = 1;
if (timer_settime(timer, 0, &itimerspec, NULL) < 0)
        perror("timer_settime"), exit(1);

for(;;) pause();
```

```
              perf kvm stat record -a sleep 1

                   HOT  CAN_GET_HOT  ????

              VM-EXIT    Samples  Samples%
            MSR_WRITE     338793    56.54%
    PENDING_INTERRUPT     168431    28.11%
     PREEMPTION_TIMER      91723    15.31%
   EXTERNAL_INTERRUPT        234     0.04%
                  HLT         65     0.01%
             MSR_READ          6     0.00%
        EPT_MISCONFIG          6     0.00%
```

Hard to virtualize guest workloads

- Some databases incidentally tend to be very heavy in terms of:
    - Frequent scheduling on potentially otherwise idle vCPU
    - Programming high frequency timers running at fairly high frequency
- Even an increase of 10% in the computation time of guest mode compared to bare metal can become quite problematic
    - Every 1% lost anywhere matters if the maximum you can lose is 10%
- Performance regressed for those hard to virtualize workloads since *Jan 4 2018*
    - **“spectre-v2” default retpoline mitigation is important in the KVM host**
    - **“spec_store_bypass_disable=seccomp spectre_v2_user=seccomp” is *still* used as the guest default**

Recap: 4 different attack targets



2 HT/SMT
1 CORE

Guest Mode (user task)

User Mode (user task)

CPU 0
CORE 0
RUNQUEUE

CPU 1
CORE 0
RUNQUEUE

Guest Mode (user task)

User Mode (user task)

CPU 2
CORE 1
RUNQUEUE

CPU 3
CORE 1
RUNQUEUE

2 HT/SMT
1 CORE

Running tasks

User Mode (user task)

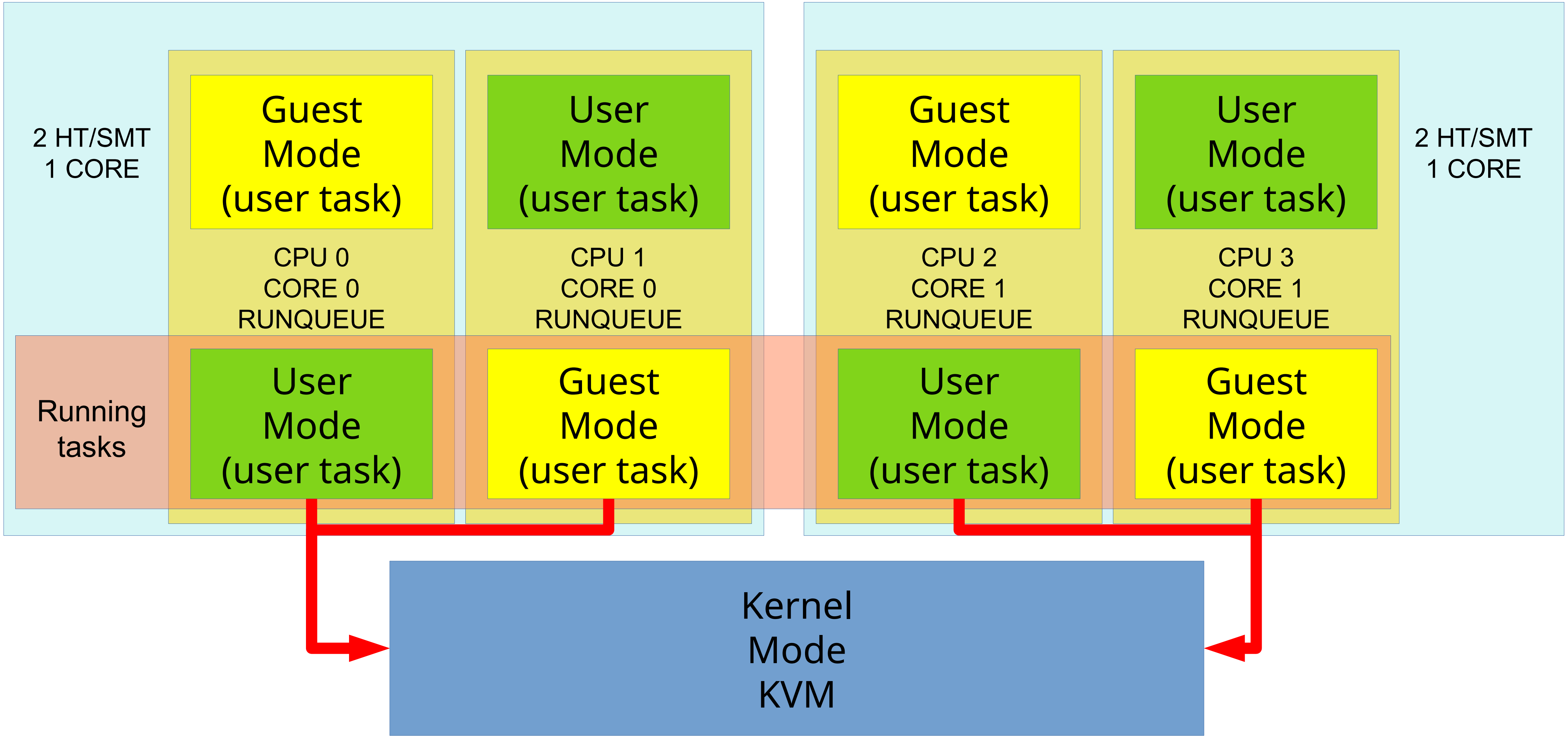
Guest Mode (user task)

User Mode (user task)

Guest Mode (user task)

Kernel Mode / KVM

Kernel attack (retpoline/IBRS/verw/PTI)



2 HT/SMT
1 CORE

Guest
Mode
(user task)

User
Mode
(user task)

Guest
Mode
(user task)

User
Mode
(user task)

2 HT/SMT
1 CORE

CPU 0
CORE 0
RUNQUEUE

CPU 1
CORE 0
RUNQUEUE

CPU 2
CORE 1
RUNQUEUE

CPU 3
CORE 1
RUNQUEUE

Running
tasks

User
Mode
(user task)

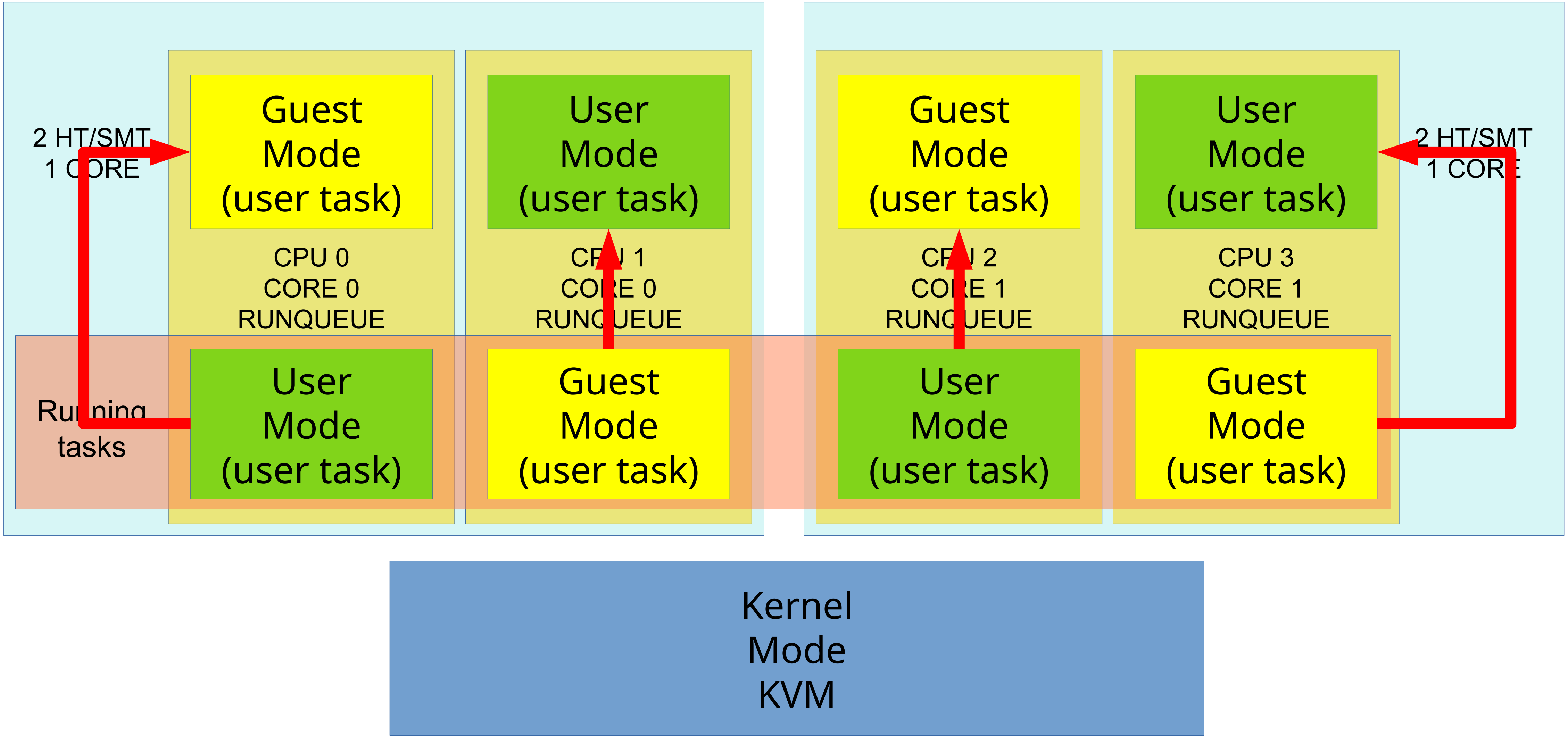
Guest
Mode
(user task)

User
Mode
(user task)

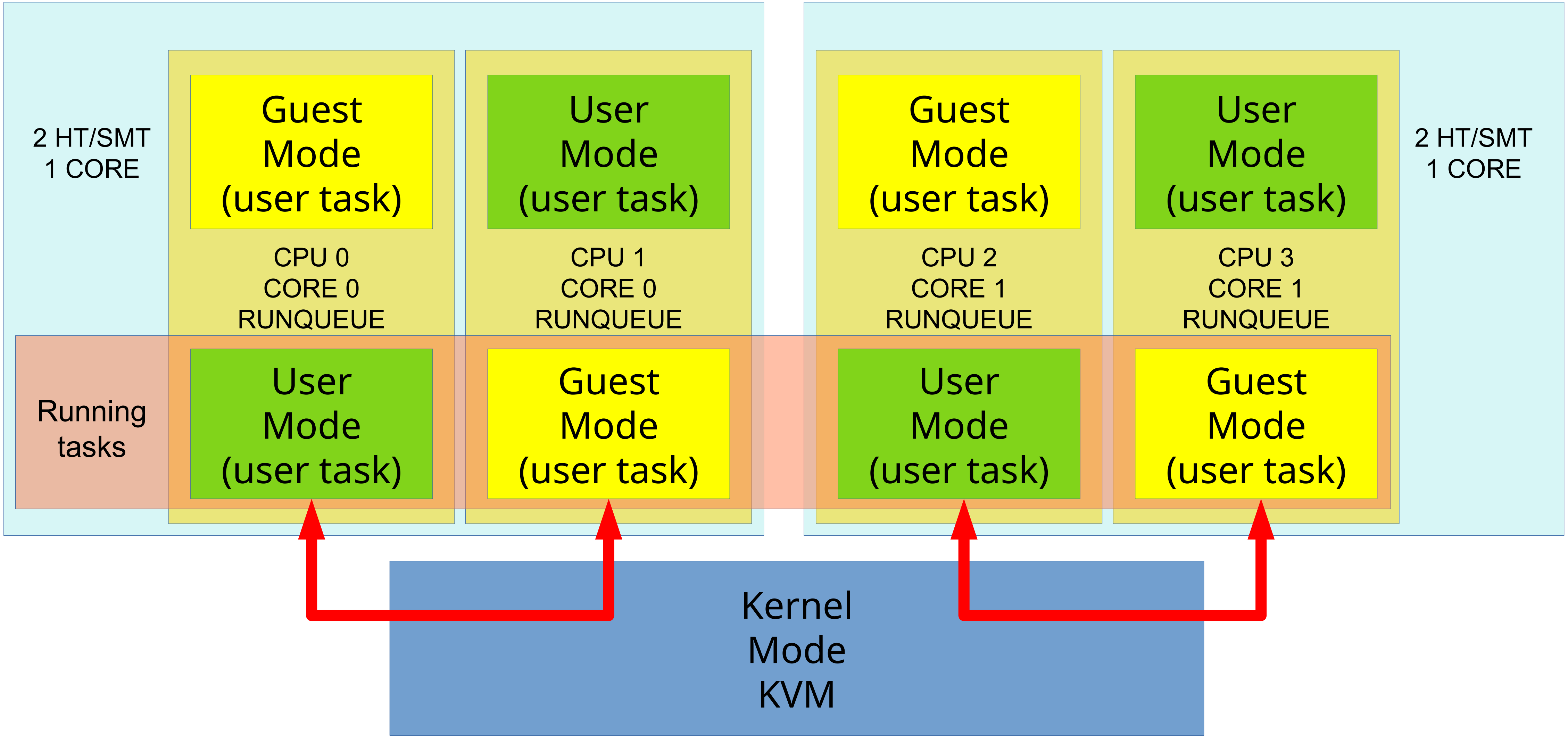
Guest
Mode
(user task)

Kernel
Mode
KVM

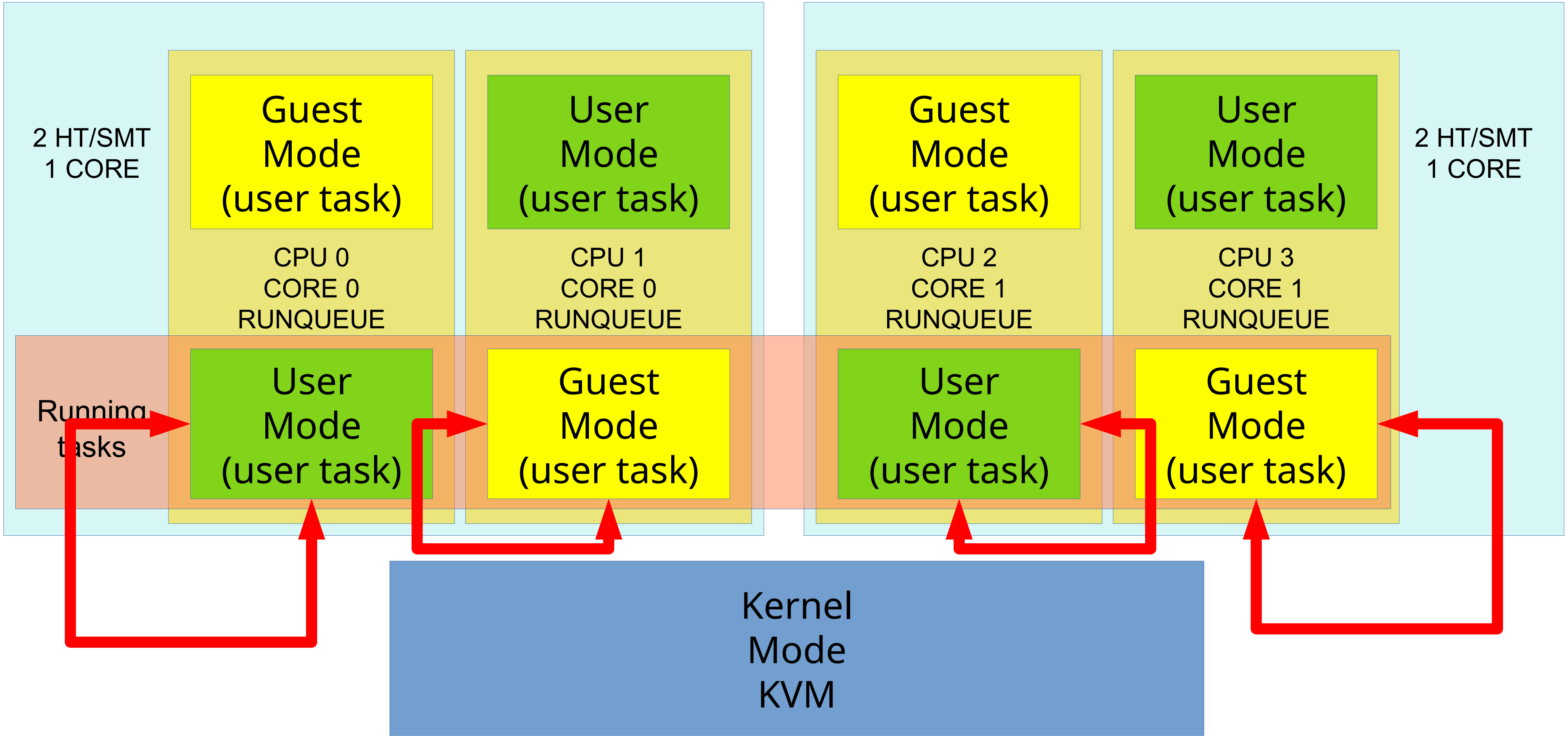
Context switch attack (IBPB/RSBfill)



HT/SMT attack (STIBP/nosmt/ASI)



Within-process JIT attack (SSBD)



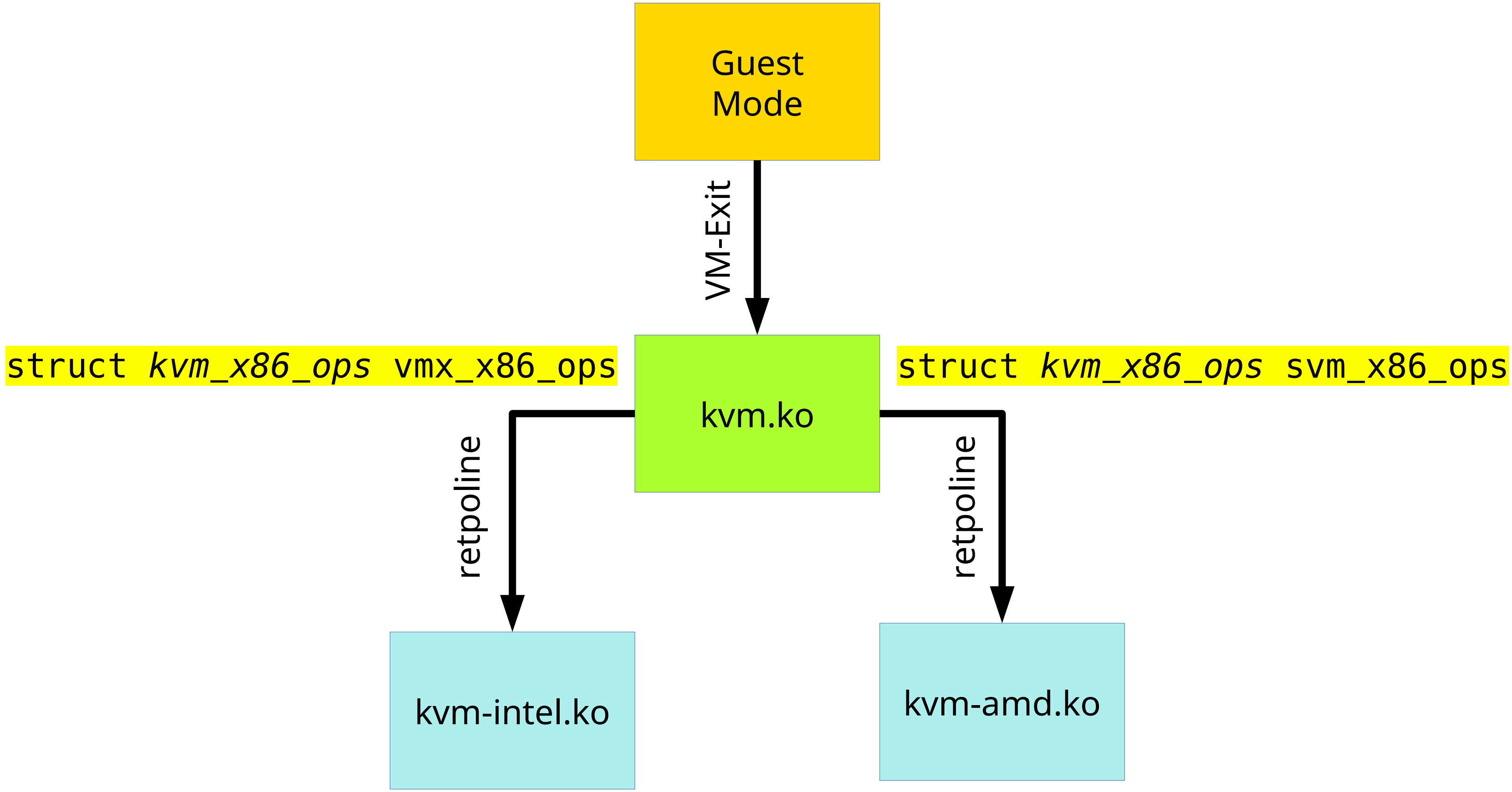
Mitigations opt-outs

- For vulnerabilities that don’t require knowing the code that is running in the CPU:
  - Meltdown → *pti=off*
  - L1TF → *l1tf=off*
  - MDS → *mds=off*
  - fpu state and other registers → no turnoff
- For vulnerabilities that require knowing the code that is running in the CPU:
  - Spectre v1 (barrier_nospec/swapgs etc..) → *nospectre_v1*
  - Spectre v2 → *spectre_v2=off* (kernel & context switch & HT attack)
  - Spectre v2 → *spectre_v2_user=off* (HT attack only)
  - SSBD → *spec_store_bypass_disable=off* (within process attack on the JIT memory from the JITed code)
- Global turnoff for all: *mitigations=off* (>= RHEL7.7)

KVM impact of spectre-v2 mitigation

- The spectre-v2 attack on the kernel/KVM by default is mitigated with ***retpolines***
- ***retpolines*** are the best performing mitigation available
  - On some CPUs it’s a full fix
  - On some CPUs “risk of an attack low”
    - On those CPUs RHEL kernels inform you in the boot log that you can opt-in the full fix with *spectre_v2=ibrs*
- ***kvm.ko*** calls ***kvm_intel.ko*** or ***kvm_amd.ko*** at every *VM-Exit* multiple times through the ***kvm_x86_ops*** pointer to functions
  - This was not optimal before, but it become slower with retpolines causing extra cost for each single invocation of the ***kvm_x86_ops*** virtual methods

KVM x86 sub-modules with *kvm_x86_ops*



hrtimer 1sec - top 10 retpolines - VMX

```
vcpu_enter_guest+772
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 397680
```

```
vcpu_enter_guest+168
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 198848
```

```
vcpu_enter_guest+486
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 198801
```

```
vcpu_enter_guest+423
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 198793
```

```
vcpu_enter_guest+575
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 198771
```

```
vmx_vcpu_run.part.88+358
vcpu_enter_guest+423
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 198736
```

```
vcpu_enter_guest+1689
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 197697
```

```
vcpu_enter_guest+4009
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 132405
```

```
skip_emulated_instruction+48
kvm_skip_emulated_instruction+82
handle_wrmsr+102
vcpu_enter_guest+772
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 131046
```

```
handle_wrmsr+85
vcpu_enter_guest+772
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 131043
```

hrtimer 1sec - top 10 retpolines – SVM

```
vcpu_enter_guest+772
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 227076
```

```
vcpu_enter_guest+168
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 113601
```

```
vcpu_enter_guest+486
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 113414
```

```
vcpu_enter_guest+423
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 113386
```

```
vcpu_enter_guest+575
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 113371
```

```
vcpu_enter_guest+1689
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 112579
```

```
vcpu_enter_guest+4009
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 75812
```

```
kvm_get_rflags+28
svm_interrupt_allowed+50
vcpu_enter_guest+4009
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 75647
```

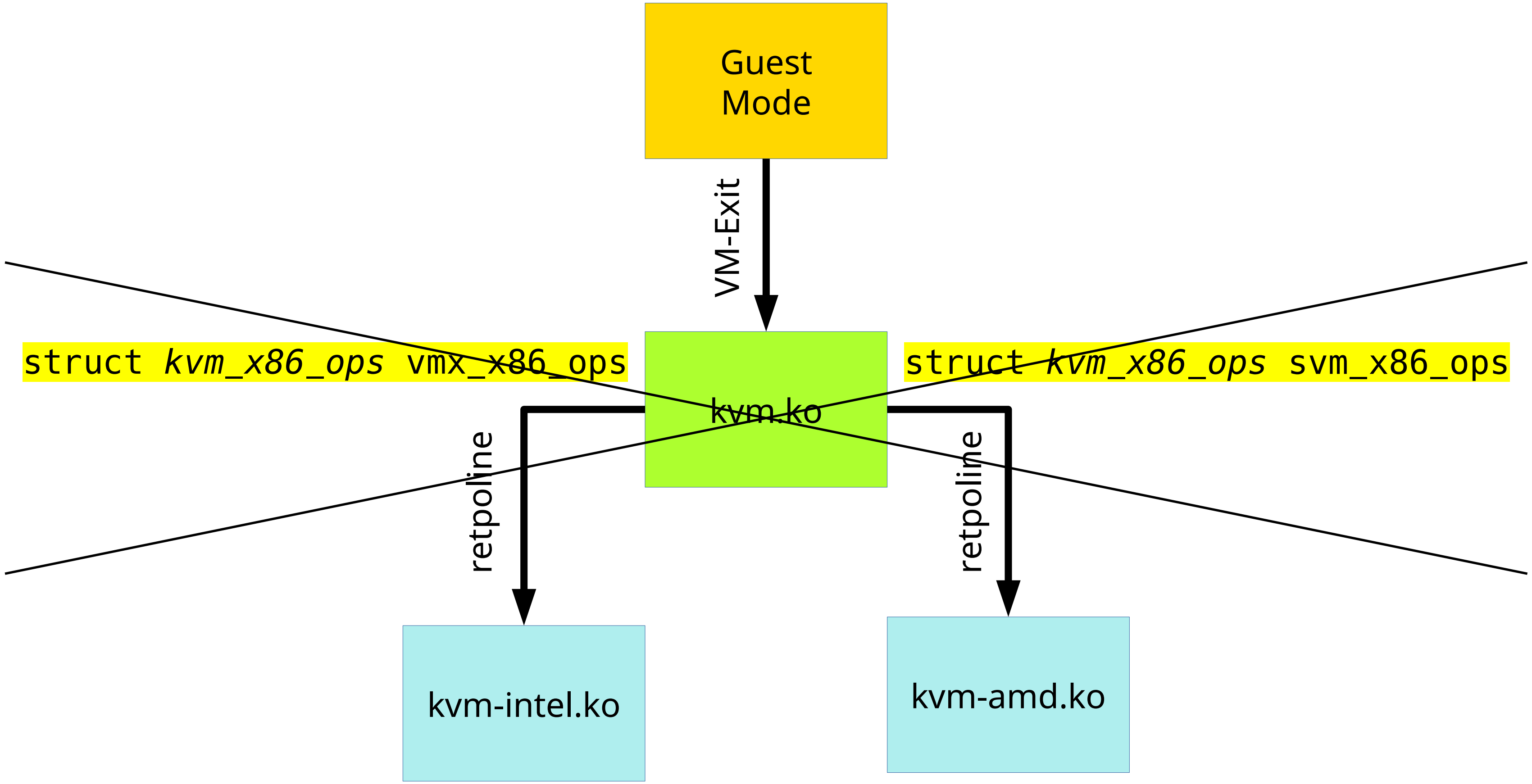
```
msr_interception+138
vcpu_enter_guest+772
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 74795
```

```
kvm_skip_emulated_instruction+49
msr_interception+356
vcpu_enter_guest+772
kvm_arch_vcpu_ioctl_run+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 74757
```

KVM monolithic

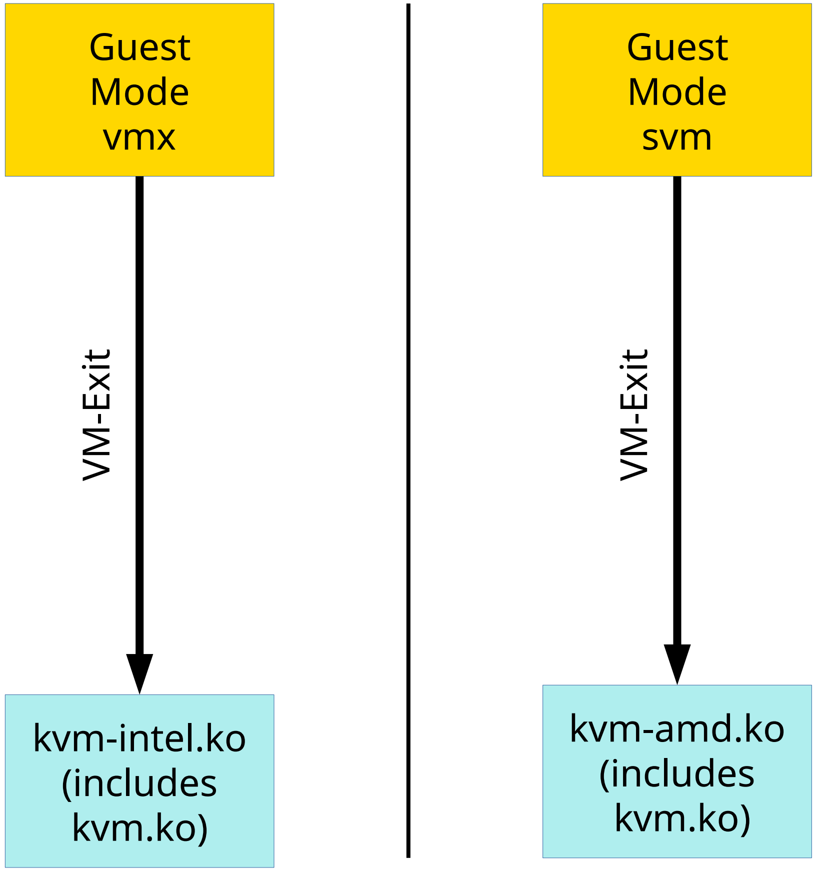
- The objective is the retpoline elimination from VM-Exits
  - remove the ***`kvm_x86_ops`***
    - remove ***kvm.ko***
      - the executable ***.o objects*** previously linked in ***kvm.ko*** need to be duplicated and linked statically in **both** ***kvm-intel.ko*** and ***kvm-amd.ko***
- pvops could eliminate retpolines, but they're suboptimal for iTLB (and RAM) costs
- Only two cons depending on CONFIG_KVM_INTEL and CONFIG_KVM_AMD .config:
  - Only one of the two options can be set to "***=y" at once***
    - Hint: distro kernels sets both "***=m"***
  - If both set ***"=m"***, a ***few MiB of disk space*** will be lost in */lib/modules/*

KVM x86 sub-modules with *kvm_x86_ops*



- No benefit: kvm-intel.ko and kvm-amd.ko can’t be loaded at the same time
  - Because of hardware constraints

KVM monolithic (no *kvm_x86_ops*)



- Replace all *kvm_x86_ops* methods with external calls with the same name, but implemented differently in kvm-intel.ko and kvm-amd.ko
- Link all kvm.ko code into both kvm-intel.ko and kvm-amd.ko

KVM VM-Exit handler optimization

- ***`kvm_x86_ops`*** (and ***`kvm_pmu_ops`***) aren't the only sources of frequent ***retpolines*** during VM-Exits
- Unlike the ***`kvm_x86_ops`***, invoking the VM-Exit reason handler *pointer to function* was optimal if the retpolines are not enabled
  - In this case we can add the retpoline optimization conditional to `#ifdef CONFIG_RETPOLINE`

KVM VM-Exit handler optimization - VMX

```
        if (exit_reason < kvm_vmx_max_exit_handlers
-           && kvm_vmx_exit_handlers[exit_reason])
+           && kvm_vmx_exit_handlers[exit_reason]) {
+#ifdef CONFIG_RETPOLINE
+               if (exit_reason == EXIT_REASON_MSR_WRITE)
+                       return kvm_emulate_wrmsr(vcpu);
+               else if (exit_reason == EXIT_REASON_PREEMPTION_TIMER)
+                       return handle_preemption_timer(vcpu);
+               else if (exit_reason == EXIT_REASON_PENDING_INTERRUPT)
+                       return handle_interrupt_window(vcpu);
+               else if (exit_reason == EXIT_REASON_EXTERNAL_INTERRUPT)
+                       return handle_external_interrupt(vcpu);
+               else if (exit_reason == EXIT_REASON_HLT)
+                       return kvm_emulate_halt(vcpu);
+               else if (exit_reason == EXIT_REASON_PAUSE_INSTRUCTION)
+                       return handle_pause(vcpu);
+               else if (exit_reason == ~~EXIT_REASON_MSR_READ~~)
+                       return kvm_emulate_rdmsr(vcpu);
+               else if (exit_reason == ~~EXIT_REASON_CPUID~~)
+                       return kvm_emulate_cpuid(vcpu);
+               else if (exit_reason == EXIT_REASON_EPT_MISCONFIG)
+                       return handle_ept_misconfig(vcpu);
+#endif
                return kvm_vmx_exit_handlers[exit_reason](vcpu);
```

+EXIT_REASON_VMCALL

KVM VM-Exit handler optimization - SVM

```
+#ifdef CONFIG_RETPOLINE
+       if (exit_code == SVM_EXIT_MSR)
+               return msr_interception(svm);
+       else if (exit_code == SVM_EXIT_VINTR)
+               return interrupt_window_interception(svm);
+       else if (exit_code == SVM_EXIT_INTR)
+               return intr_interception(svm);
+       else if (exit_code == SVM_EXIT_HLT)
+               return halt_interception(svm);
+       else if (exit_code == SVM_EXIT_NPF)
+               return npf_interception(svm);
+       else if (exit_code == ~~SVM_EXIT_CPUID~~)
+               return cpuid_interception(svm);
+#endif
        return svm_exit_handlers[exit_code](svm);
```

hrtimer 1sec - top 5 retpolines – *VMX* - after

```
__kvm_wait_lapic_expire+284
vmx_vcpu_run.part.97+1091
vcpu_enter_guest+377
kvm_arch_vcpu_ioctl_run+261
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 2390
@[]: 33410
```

```
do_syscall_64+89
]: 267
```

```
finish_task_switch+371
__schedule+573
preempt_schedule_common+10
_cond_resched+29
kvm_arch_vcpu_ioctl_run+401
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 103
```

```
__schedule+1081
preempt_schedule_common+10
_cond_resched+29
kvm_arch_vcpu_ioctl_run+401
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 103
```

```
ktime_get_update_offsets_now+70
hrtimer_interrupt+131
smp_apic_timer_interrupt+106
apic_timer_interrupt+15
vcpu_enter_guest+1119
kvm_arch_vcpu_ioctl_run+261
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 57
```

```
delay_fn()
__delay
```

hrtimer 1sec - top 5 retpolines – *SVM* - after

```
ktime_get+58
start_sw_timer+279
restart_apic_timer+85
kvm_set_msr_common+1497
msr_interception+142
vcpu_enter_guest+684
kvm_arch_vcpu_ioctl_run+261
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 499845
```

`ktime_get()`

```
ktime_get+58
clockevents_program_event+84
hrtimer_try_to_cancel+168
hrtimer_cancel+21
kvm_set_lapic_tscdeadline_msr+43
kvm_set_msr_common+1497
msr_interception+142
vcpu_enter_guest+684
kvm_arch_vcpu_ioctl_run+261
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 42848
```

`ktime_get()`

```
clockevents_program_event+148
hrtimer_try_to_cancel+168
hrtimer_cancel+21
kvm_set_lapic_tscdeadline_msr+43
kvm_set_msr_common+1497
msr_interception+142
vcpu_enter_guest+684
kvm_arch_vcpu_ioctl_run+261
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 42766
```

```
lapic_next_event+28
clockevents_program_event+148
hrtimer_try_to_cancel+168
hrtimer_cancel+21
kvm_set_lapic_tscdeadline_msr+43
kvm_set_msr_common+1497
msr_interception+142
vcpu_enter_guest+684
kvm_arch_vcpu_ioctl_run+261
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 42723
```

`apic->write();`

```
ktime_get+58
clockevents_program_event+84
hrtimer_start_range_ns+528
start_sw_timer+356
restart_apic_timer+85
kvm_set_msr_common+1497
msr_interception+142
vcpu_enter_guest+684
kvm_arch_vcpu_ioctl_run+261
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]: 41887
```

`ktime_get()`

Micro **benchmark disclaimer**

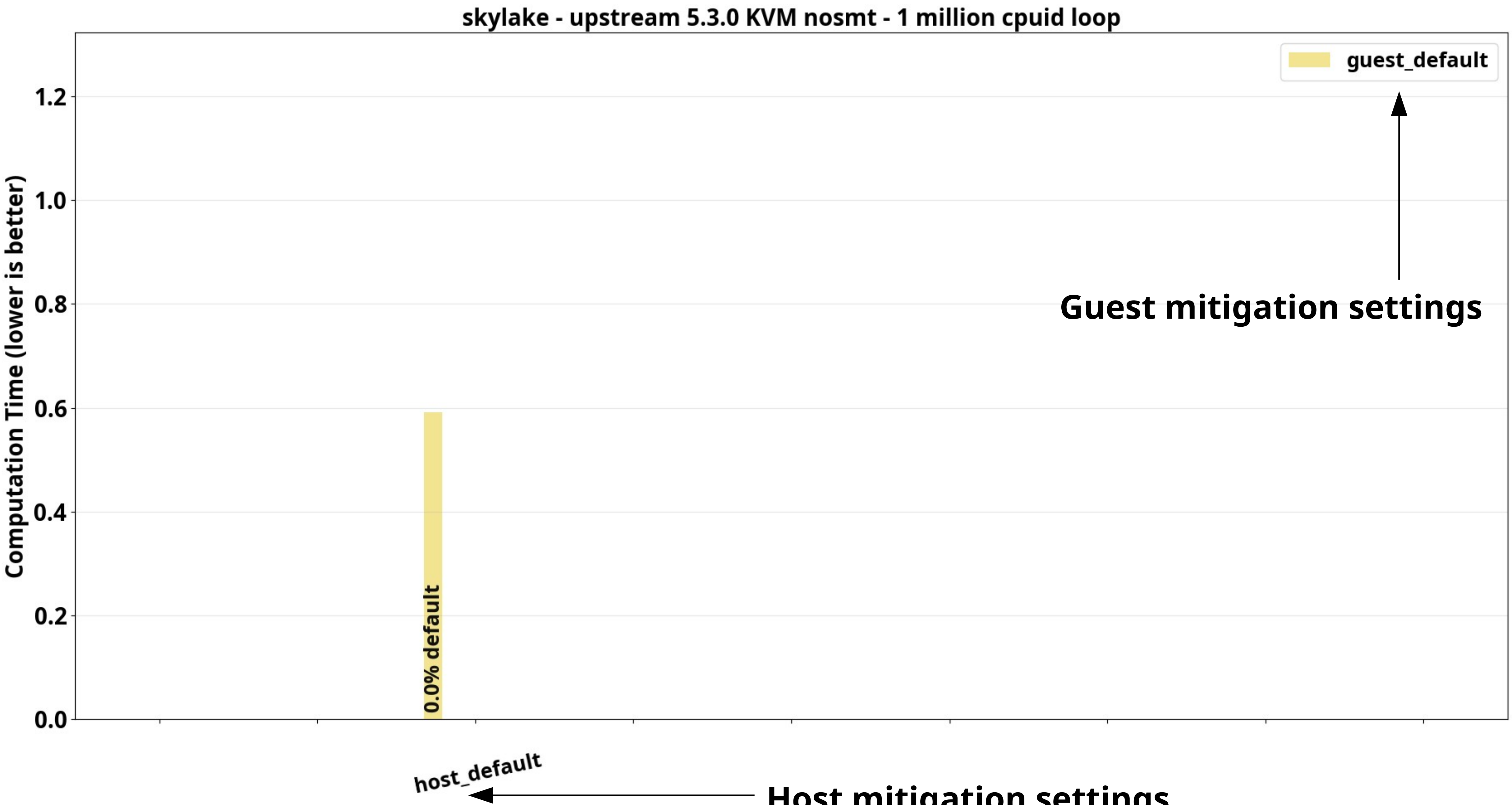
- The following slides are going to show only ***micro*** benchmarks
- All ***micro*** benchmarks are ***not representative*** of ***useful*** or ***real life workloads*** or real life software applications or real life software products
- The results shown in the next slides ***should not be taken at face value*** and they don't represent the real impact of the software mitigations against speculative execution side channel attacks
- All real and useful software applications running on Linux will show completely different benchmark results (i.e. a ***much lower impact***) than what is shown in this slide deck
- ***Micro*** benchmarks in this slide deck are provided with the only purpose of
  - explaining how the software mitigation works
  - ***justifying*** to the community the KVM monolithic optimization developments or/and equivalent ***software optimizations***

1 million CPUID loop

- Only useful to measure the VM-Exit latency

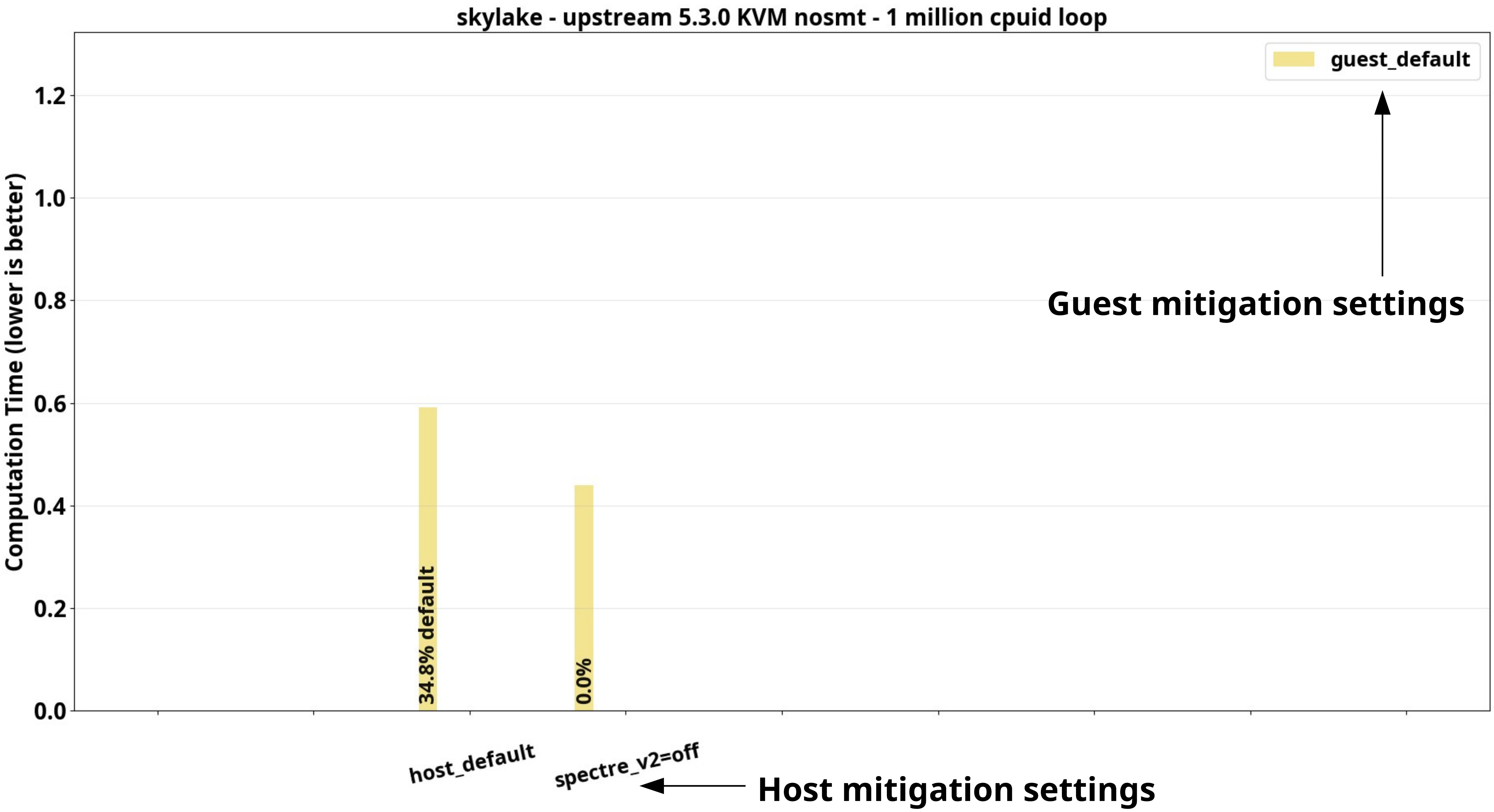
```
for (i=0; i < 1000000; i++)
  asm volatile("cpuid"
      : "=a" (eax),
        "=b" (ebx),
        "=c" (ecx),
        "=d" (edx)
      : "0" (eax), "2" (ecx)
      : "memory");
```

Red Hat



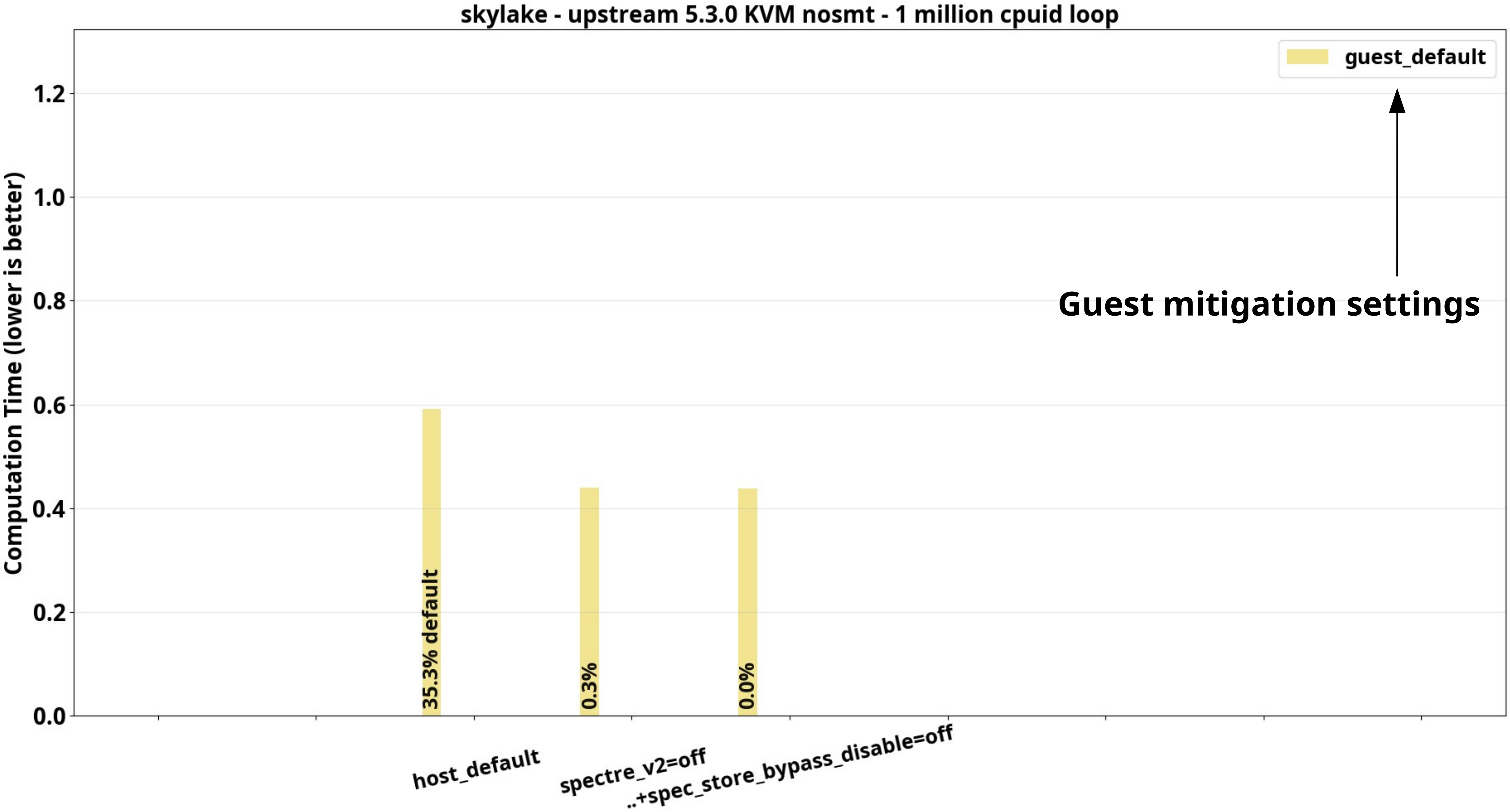
Guest mitigation settings

Host mitigation settings

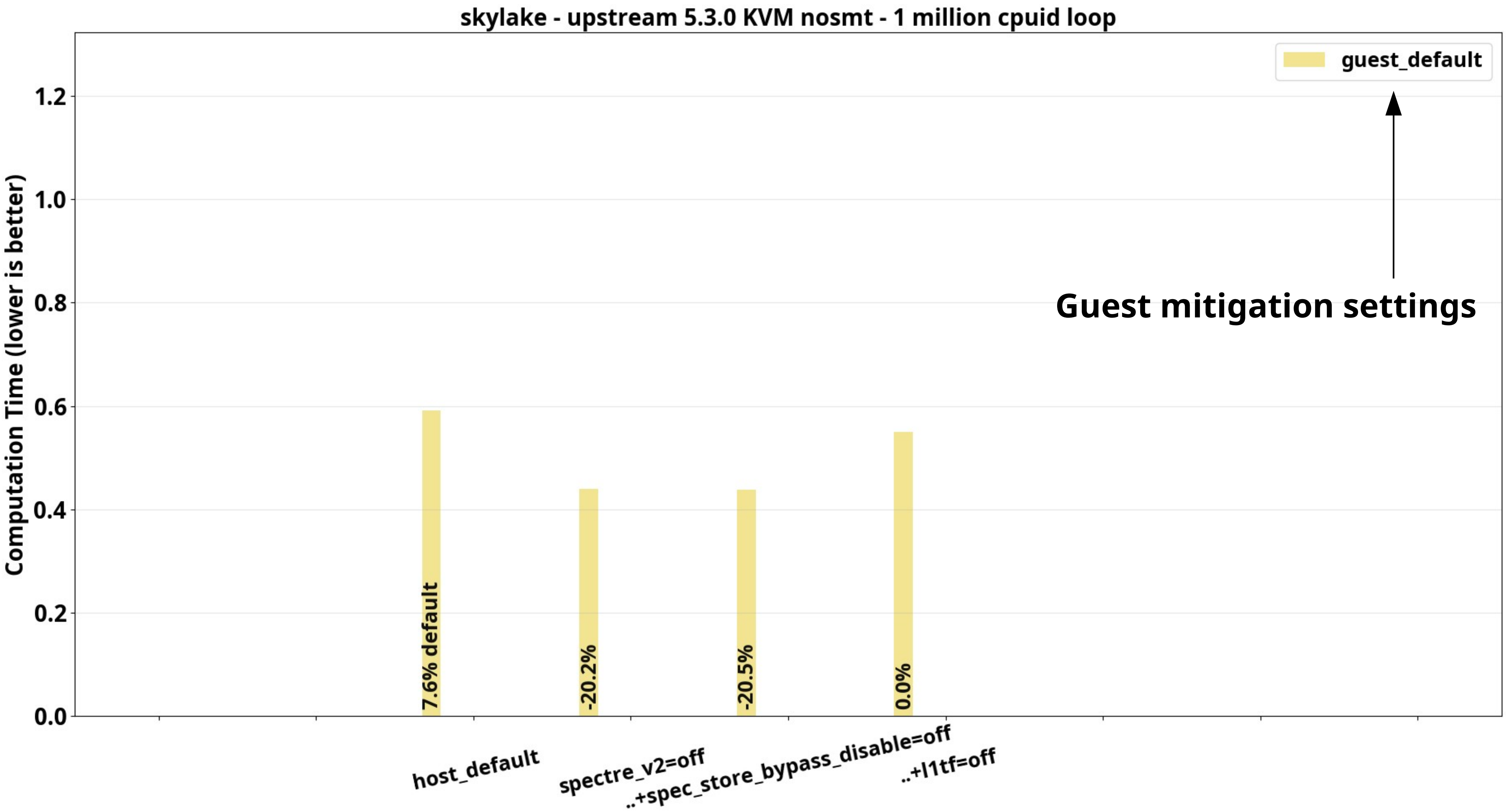


Guest mitigation settings

Host mitigation settings



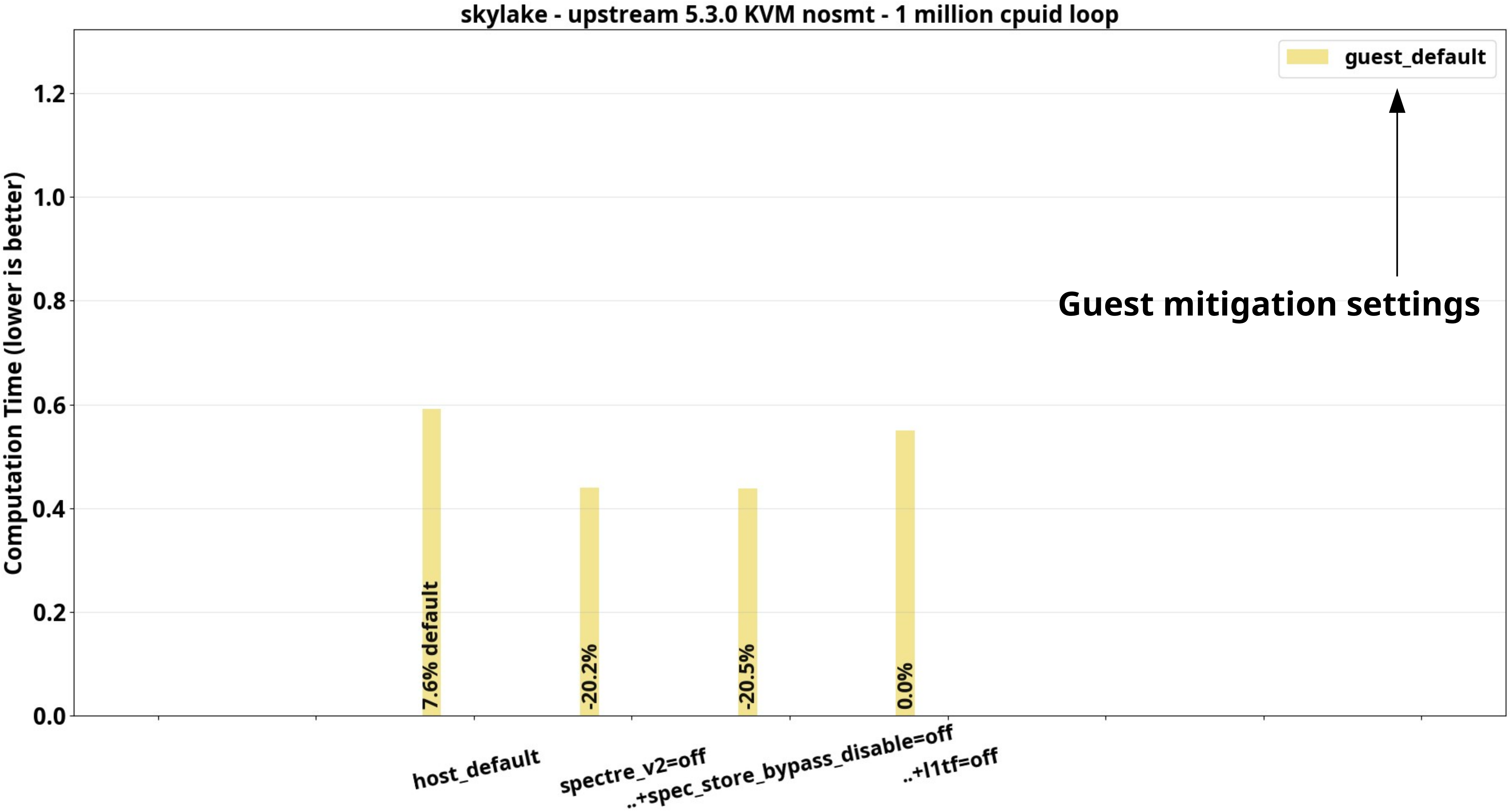
Guest mitigation settings



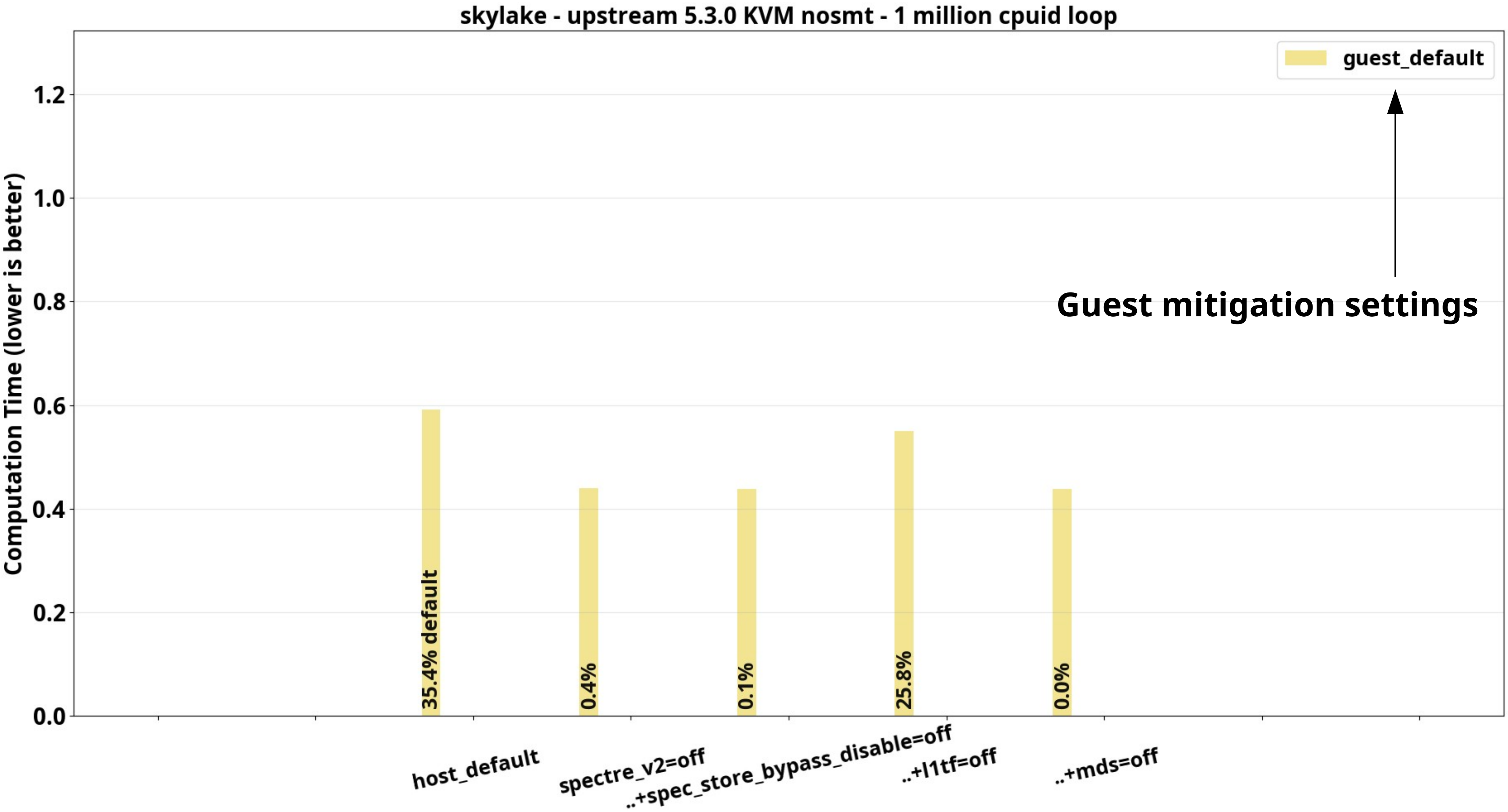
Guest mitigation settings

Why l1tf=off is slower than l1tf=flush?

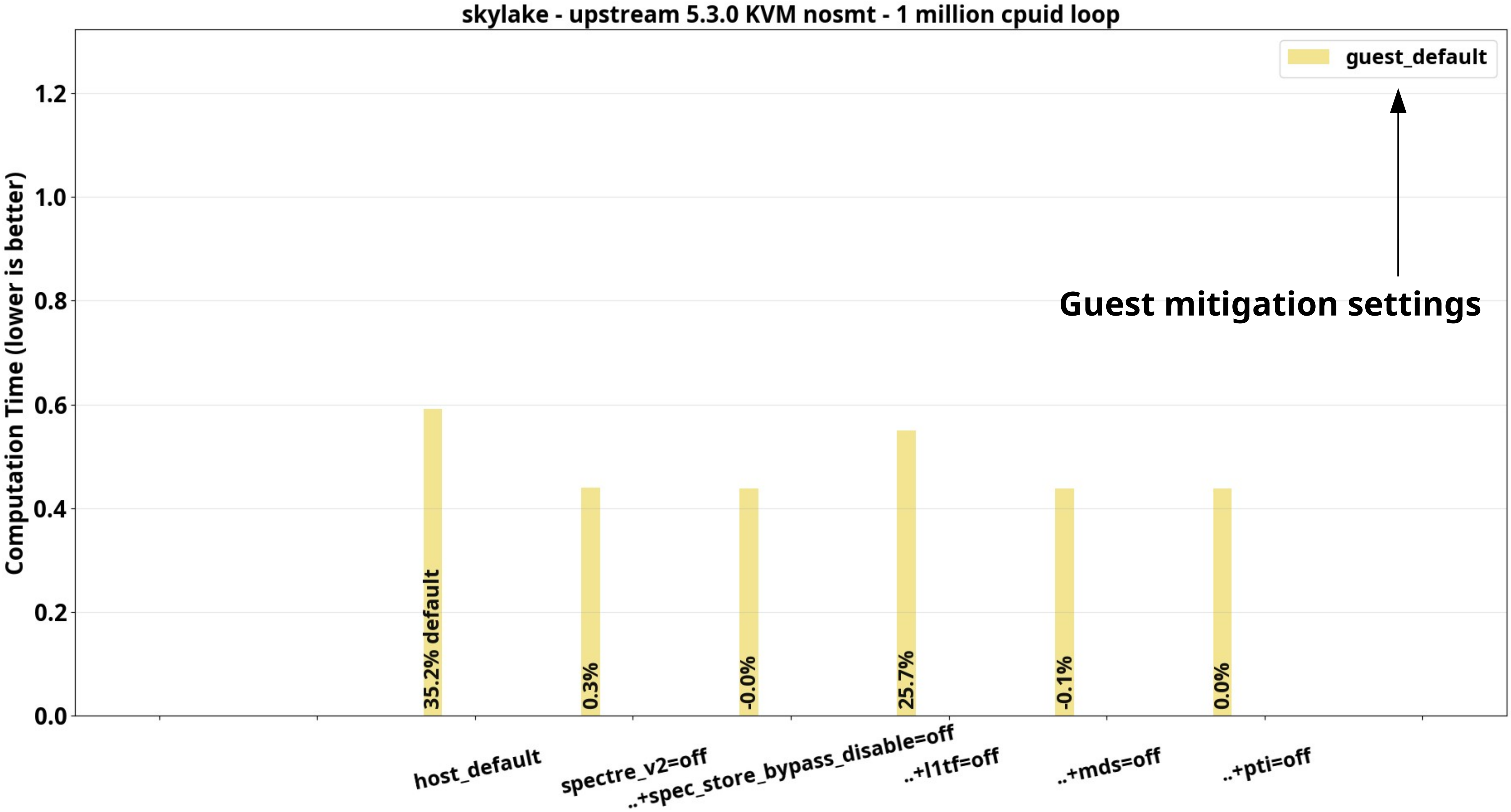
- This happens without MDS_NO and without mds=off and with MD_CLEAR set in the host cpuid
  - The *l1flush* implies *verw*, but is conditional by default:
    - ✓ After l1tf=off KVM executes verw at every VM-Enter
- CPUs with RCDL_NO always behave like l1tf=off by default
  - Should still be faster than l1tf=cond?



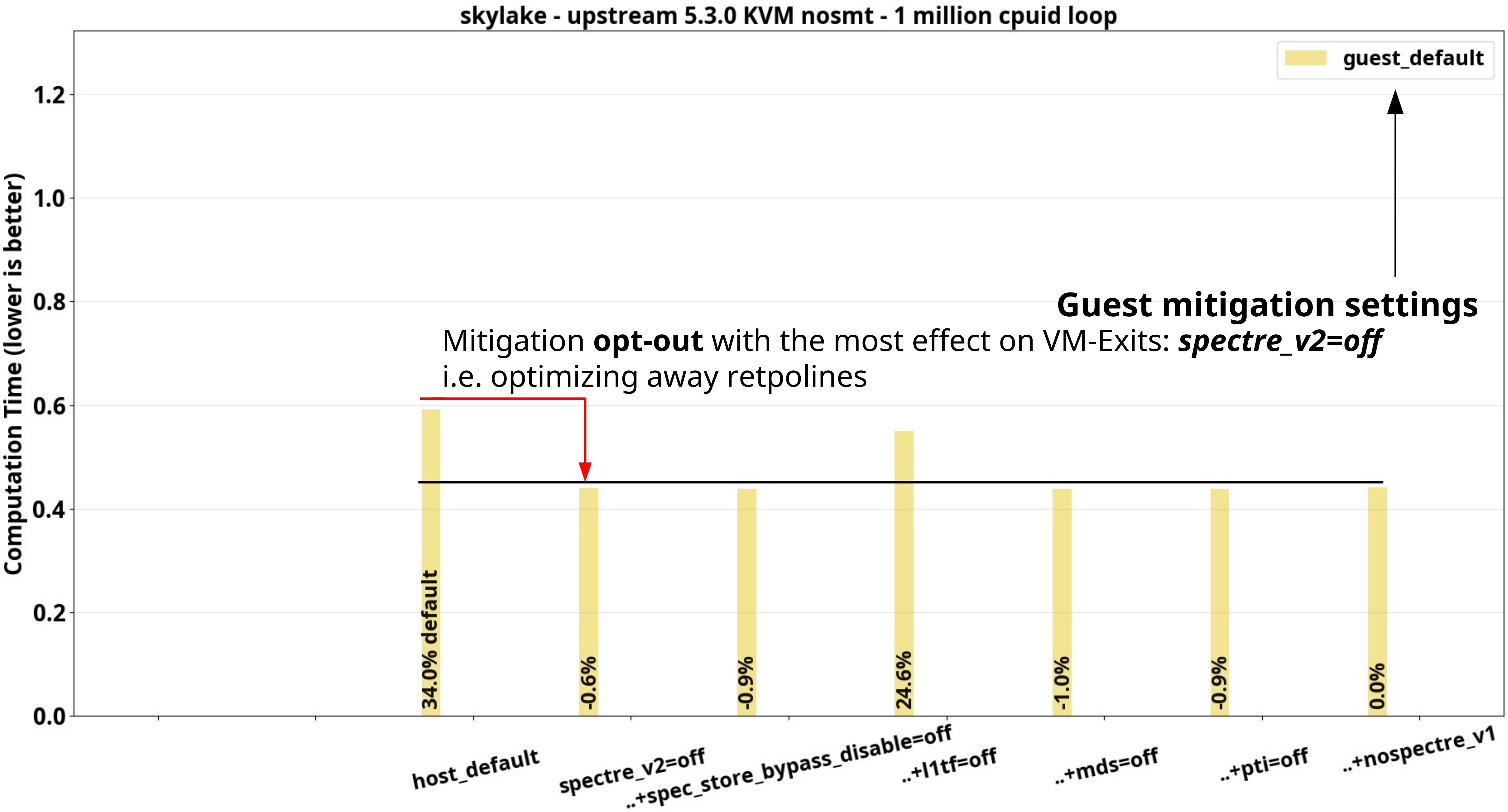
Guest mitigation settings



Guest mitigation settings

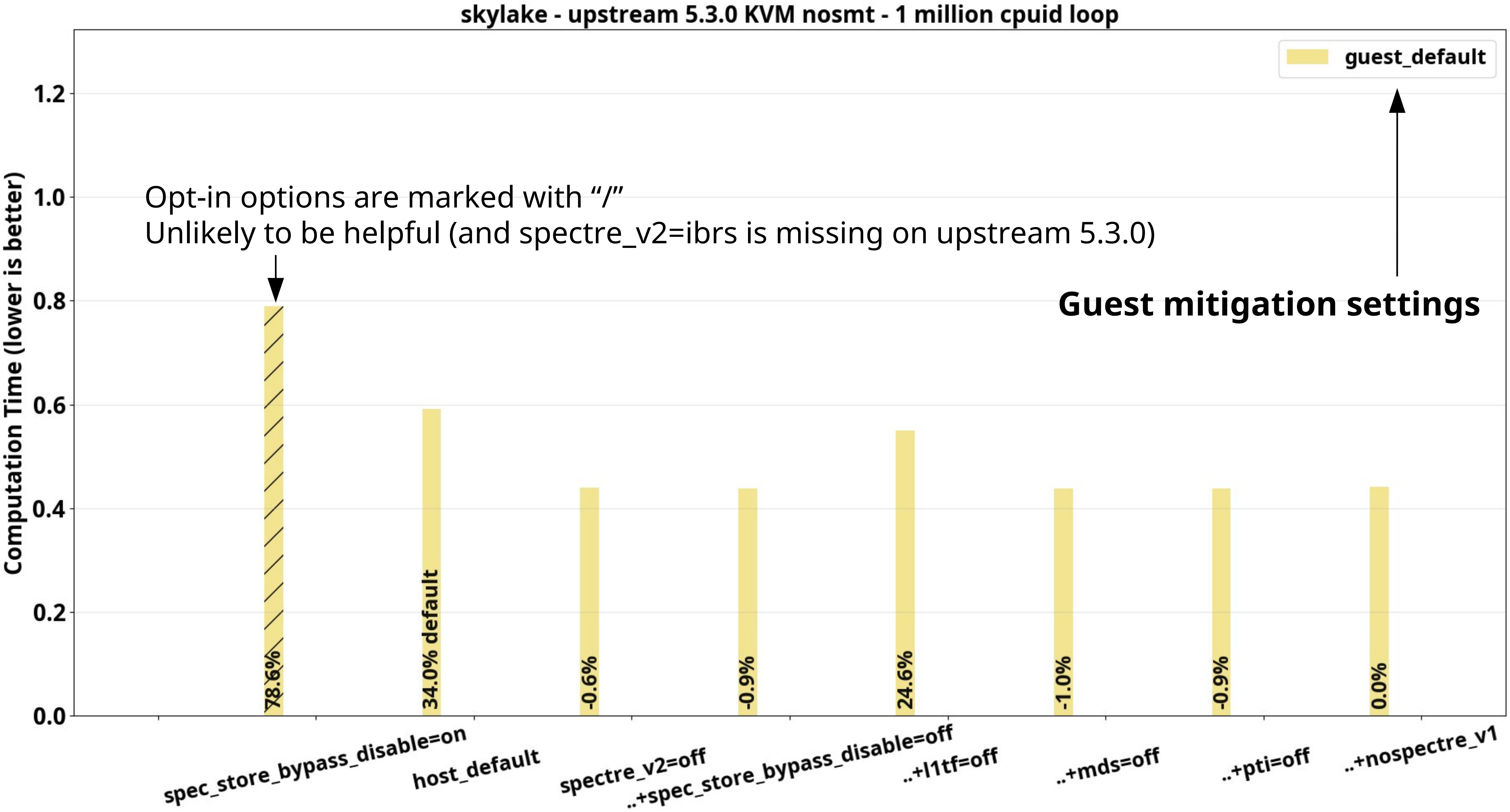


Guest mitigation settings



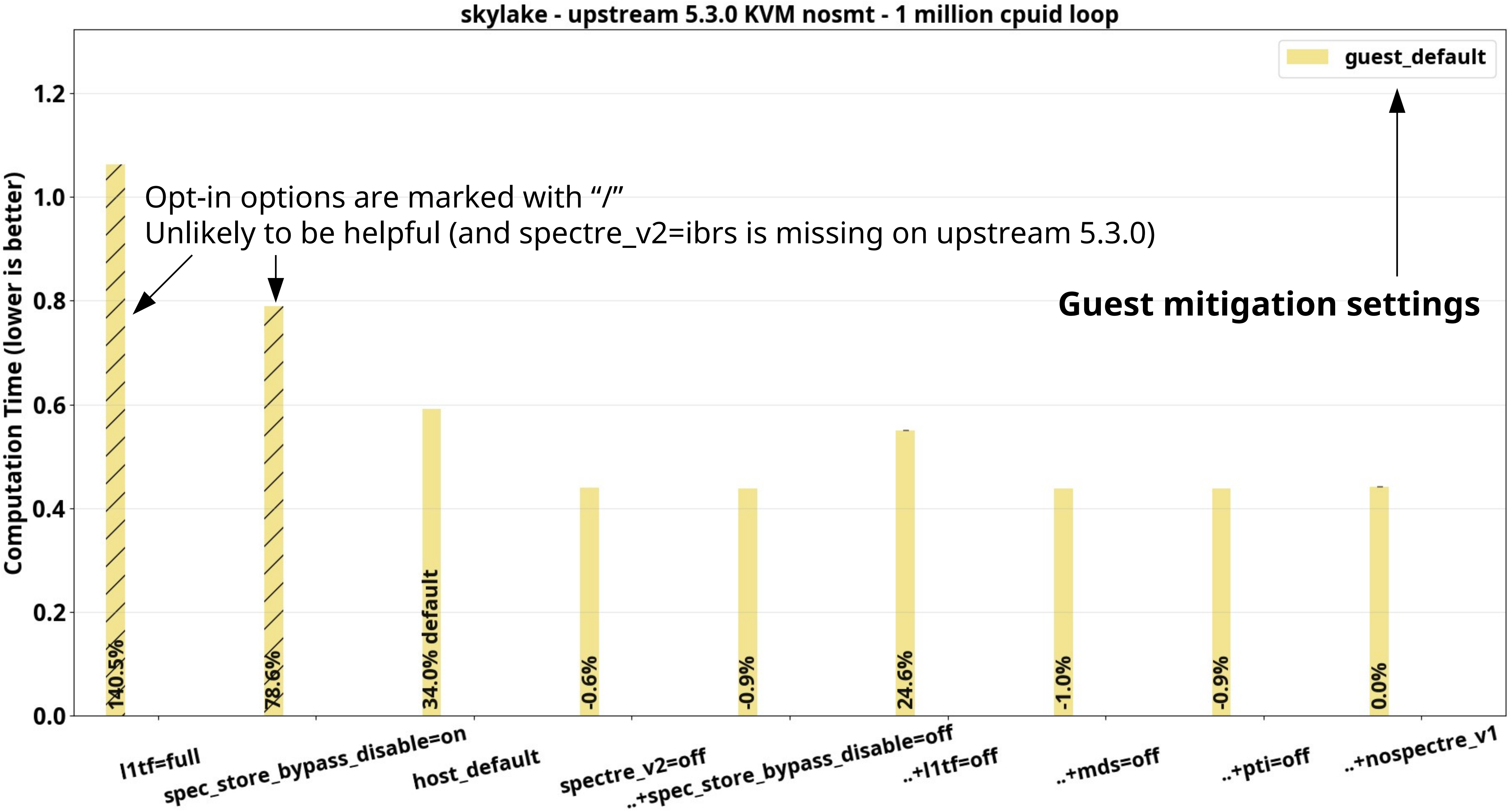
Guest mitigation settings

Mitigation **opt-out** with the most effect on VM-Exits: ***spectre_v2=off***
i.e. optimizing away retpolines



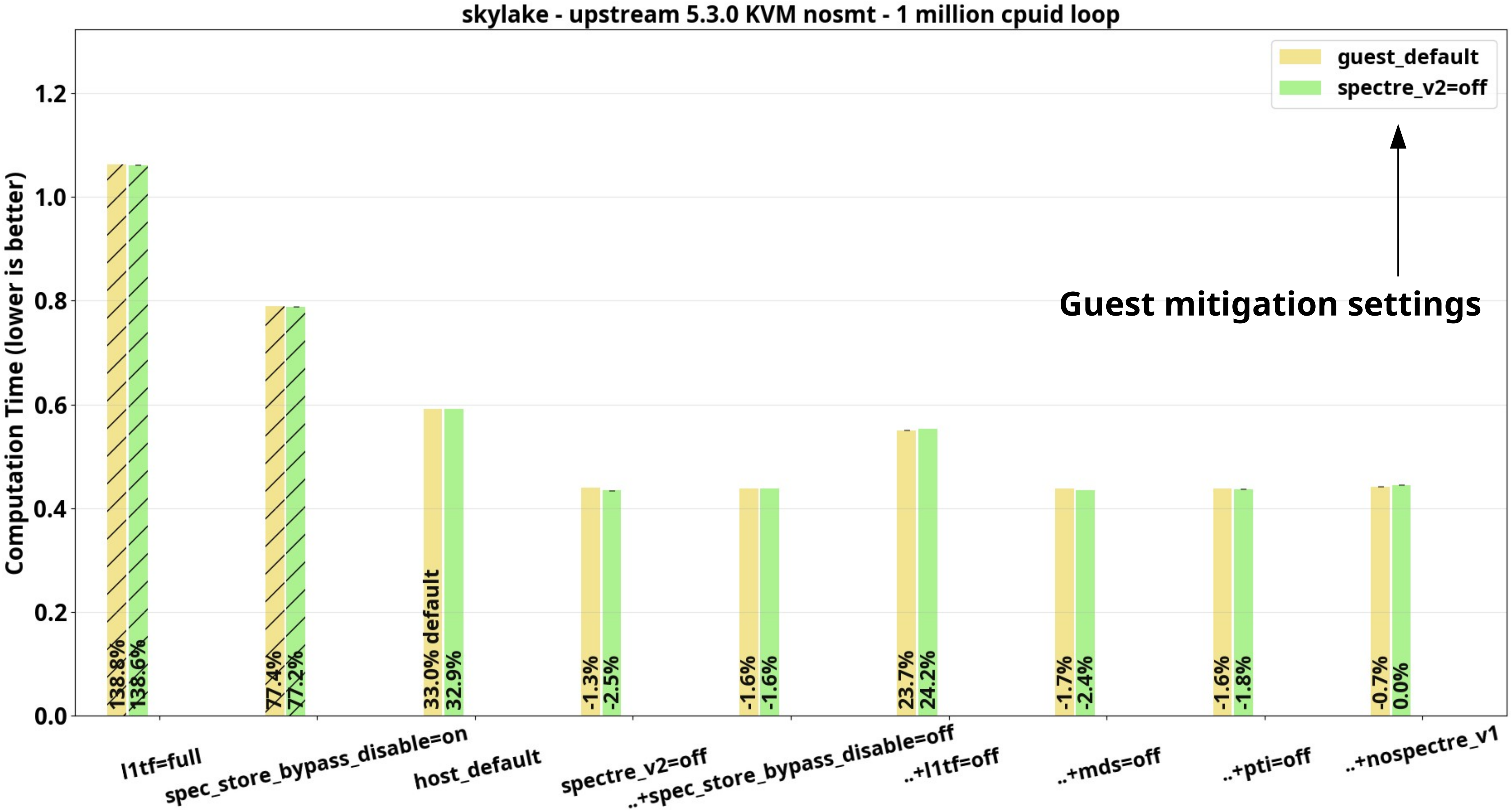
Opt-in options are marked with “/”
Unlikely to be helpful (and spectre_v2=ibrs is missing on upstream 5.3.0)

Guest mitigation settings

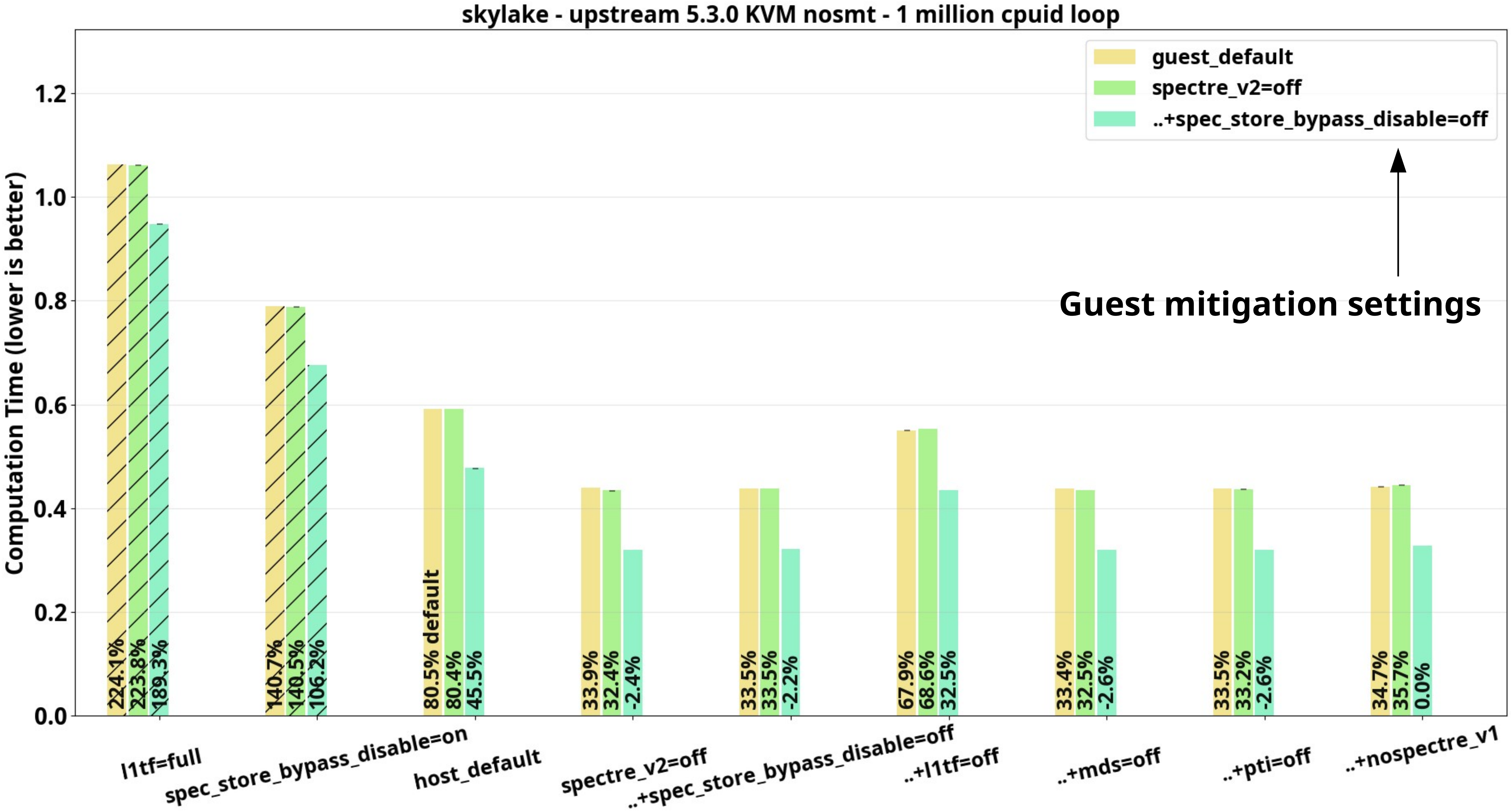


Opt-in options are marked with "/"
Unlikely to be helpful (and spectre_v2=ibrs is missing on upstream 5.3.0)

Guest mitigation settings



Guest mitigation settings



Guest mitigation settings

Why ssbd=off in guest speed up KVM?

- *spec_store_bypass_disable=auto* is the guest and host default
  - In practice the default is *spec_store_bypass_disable=seccomp*
- NOTE: the guest cpuid loop doesn’t run under SECCOMP
- Problem: nearly everything else nowadays occasionally uses SECCOMP (sshd etc..)
- The first use of SECCOMP in guest will write SSBD to **SPEC_CTRL** and ***will forever slowdown the guest***
  - ***rdmsr(SPEC_CTRL)*** forced at every VM-Exit

spectre_v2=off was already set

- *spectre_v2_user=off* was already implied by *spectre_v2=off* at the previous step
  - If not disabling *spectre_v2_user* too, **SPEC_CTRL** may still be written to for ***STIBP***
- *spectre_v2_user=auto* is the upstream default
  - With most distro .config the default is *spectre_v2_user=seccomp*

ssbd = spectre_v2_user = seccomp

- ***=seccomp*** was a good default on un-embargo day (as a "*catch-all*")
- ***=seccomp*** looks too coarse by now
  - Doesn't only hurt the guest performance
    - It hurts all SECCOMP users even on bare metal
      - More and more software runs under SECCOMP including Kubernetes pods and podman containers
- ***=prctl*** would be a preferred default now because the apps who need STIBP or SSBD should have added *`prctl(PR_SET_SPECULATION_CTRL)`*
- There has never been a ***guarantee*** that code requiring STIBP or SSBD runs under SECCOMP in the first place
- SECCOMP users are adding *`SECCOMP_FILTER_FLAG_SPEC_ALLOW`* to their userland as band-aid for the too coarse default that slowdown SECCOMP on ***bare metal***

SSBD spec_store_bypass_disable=prctl

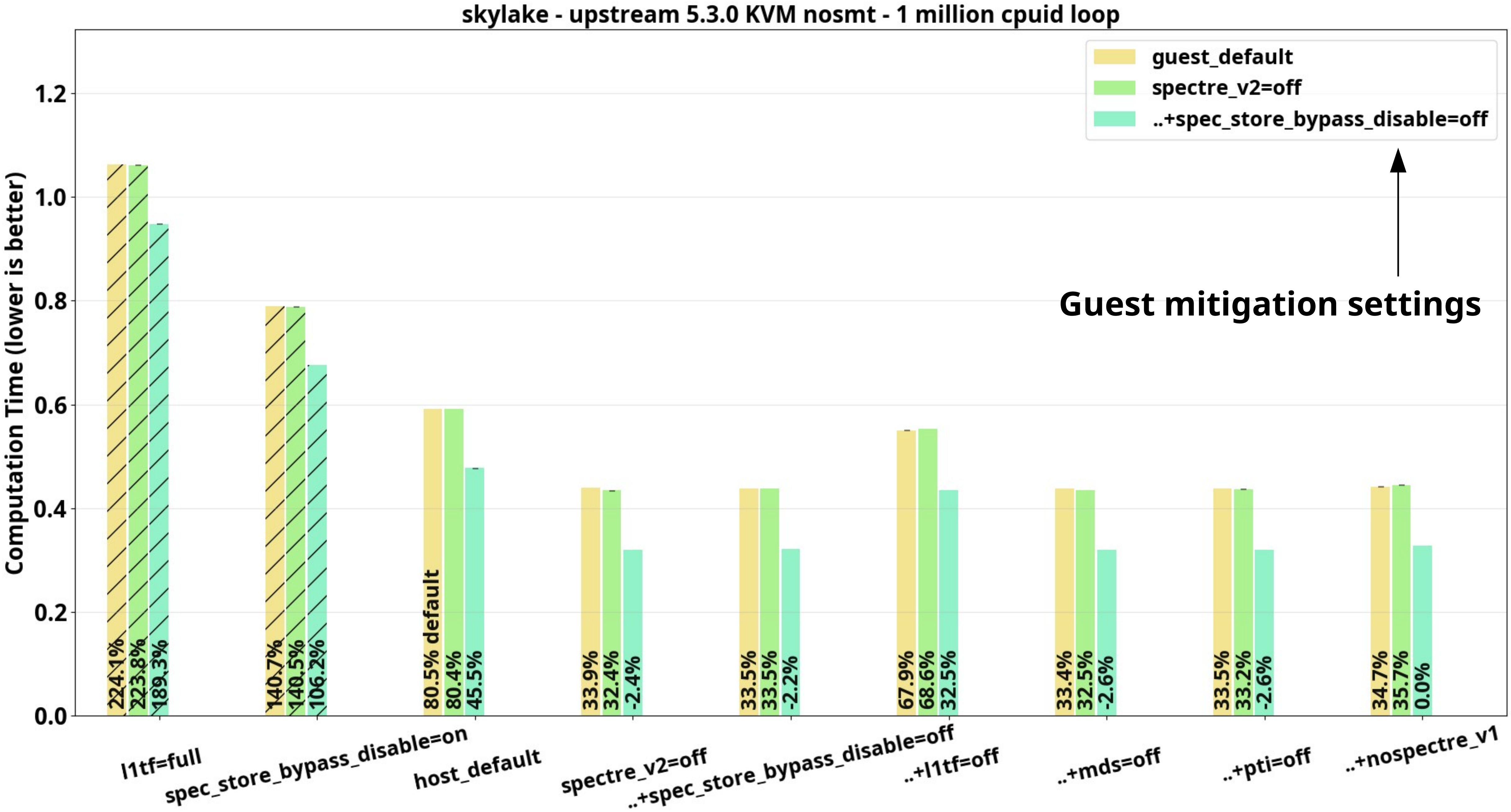
- Who really needs to set SSBD?
  - JIT running un-trusted bytecode (i.e. ***desktop*** usage of javaws/applet)
    - to avoid the JITed code to read the “in-process” memory of the JIT
- We patched OpenJDK JIT downstream with the prctl() during the SSBD embargo
- Upstream OpenJDK makes no guarantees of "in-process" data confidentiality
  - The prctl() was never submitted to OpenJDK
    - If not even the JIT enforces SSBD, why all non-JIT SECCOMP users should?
- Overall the prctl() should be worth it if the javaws/applet runs with reduced permissions
- SSBD provides no benefits after a privilege escalation that takes over the code running in any SECCOMP jail
  - After privilege escalation any malicious code can read the memory of the thread regardless if SSBD is set or not
  - Bad fit for the SECCOMP model

STIBP spectre_v2_user=prctl

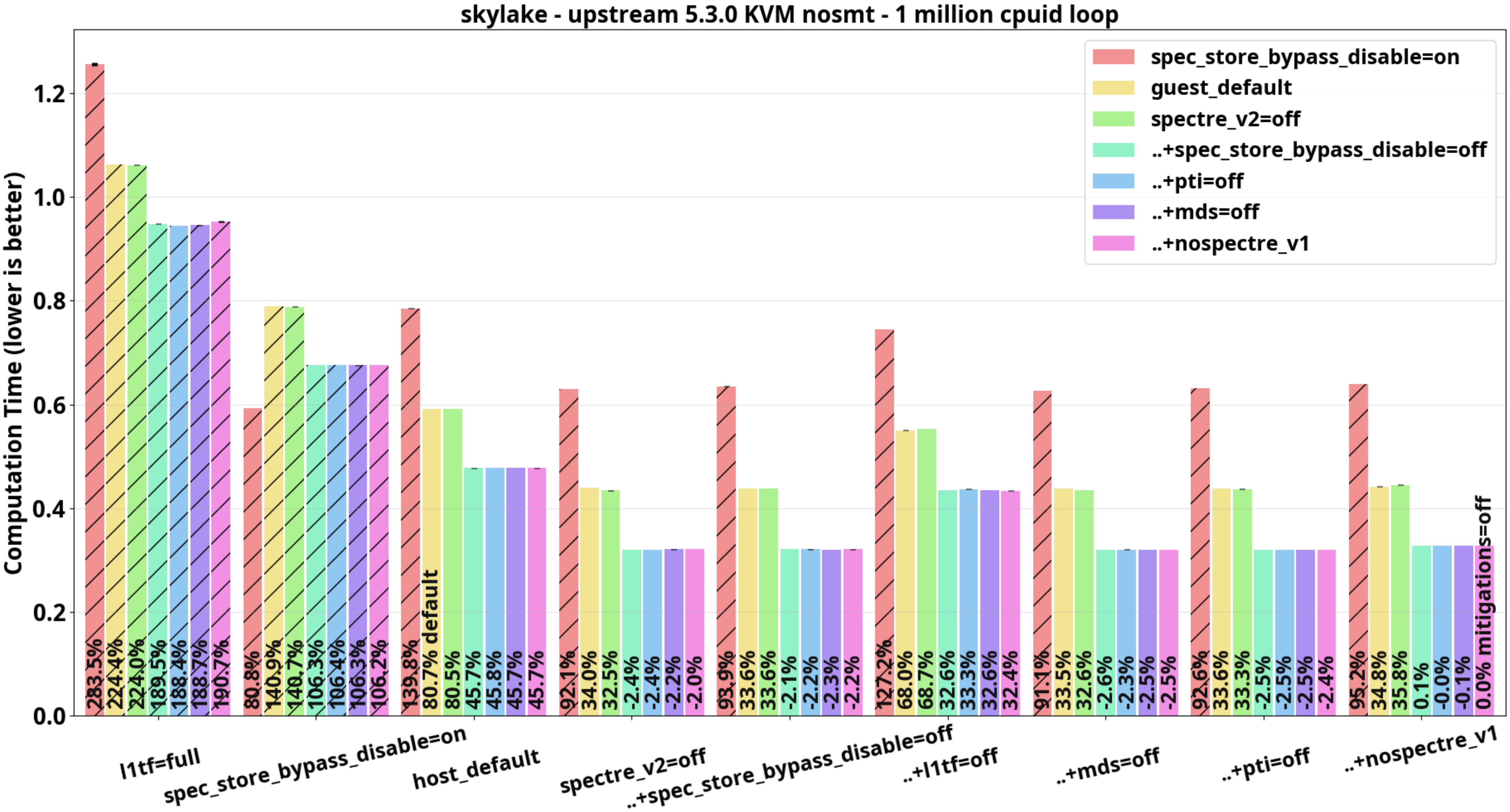
- In theory STIBP is a good fit for the SECCOMP model
- In practice the spectre-v2 HT attack prevented by STIBP would require to know ***1) the code*** and ***2) the virtual address*** it is running at in the other hyper-thread
- Security sensitive code jailed under SECCOMP would better run also under ***PID namespaces/VM isolation***
  - If so the jailed code ***can't know what's running*** in the other hyper-thread
- Even without PID namespaces/VM isolation the address is it running at is ***randomized with ASLR***

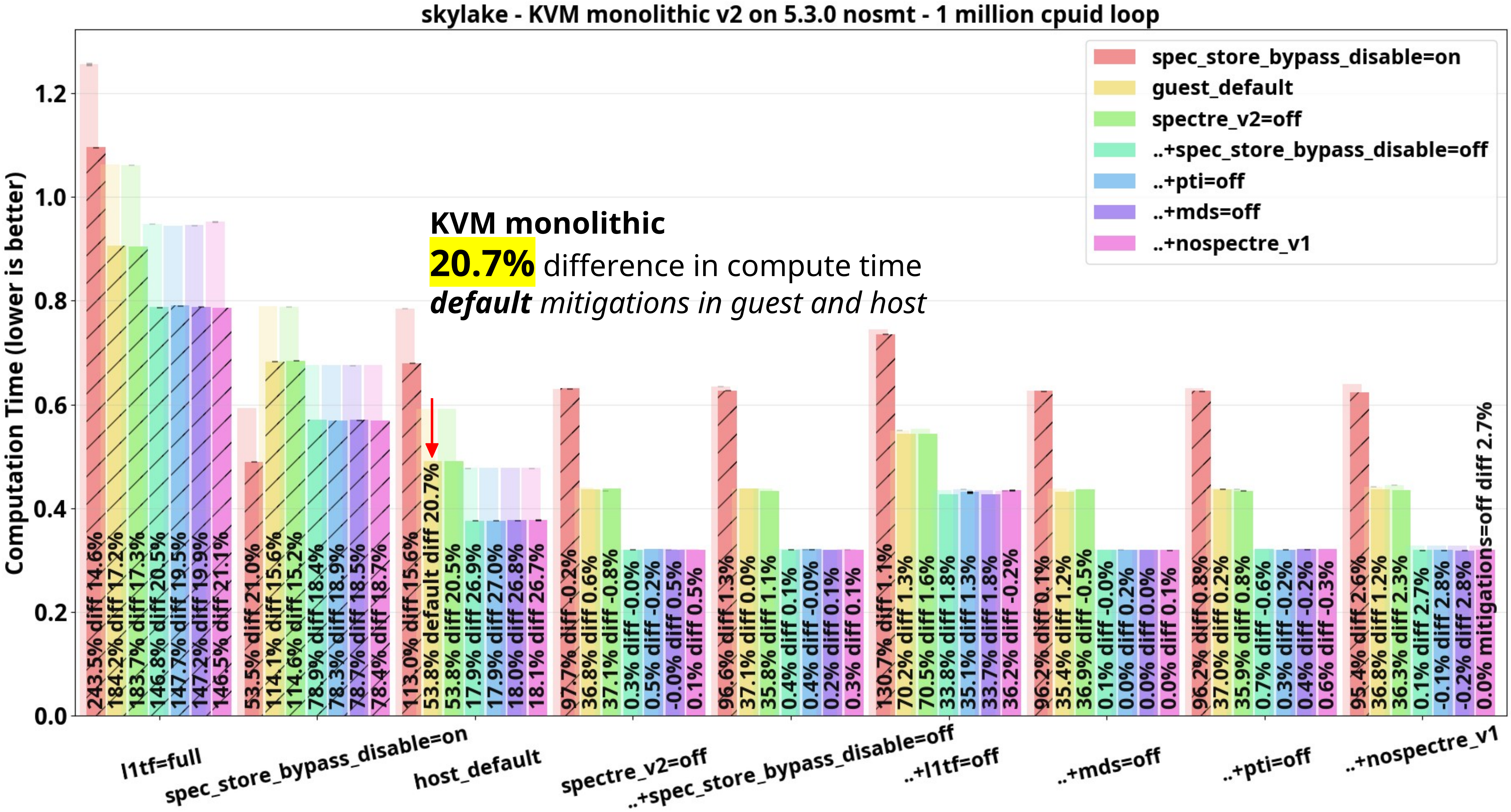
STIBP spectre_v2_user=prctl

- **Since MDS** very few CPUs are immune from MDS without ***nosmt***
  - Which would render **STIBP irrelevant** by disabling HT
- MDS can be attacked with HT enabled even without knowing what’s running in the other CPU and the virtual address it is running at
  - PID namespaces and ASLR won’t help with MDS
- STIBP mitigates ***spectre-v2-HT*** from a SECCOMP jail that **could still be able to exploit MDS**
- Even retpolines aren’t a full fix on some CPU, yet they’re the default upstream
  - A spectre-v2 attack against the ***kernel*** would have a more spread spectrum than a spectre-v2 attack against ***HT***



Guest mitigation settings

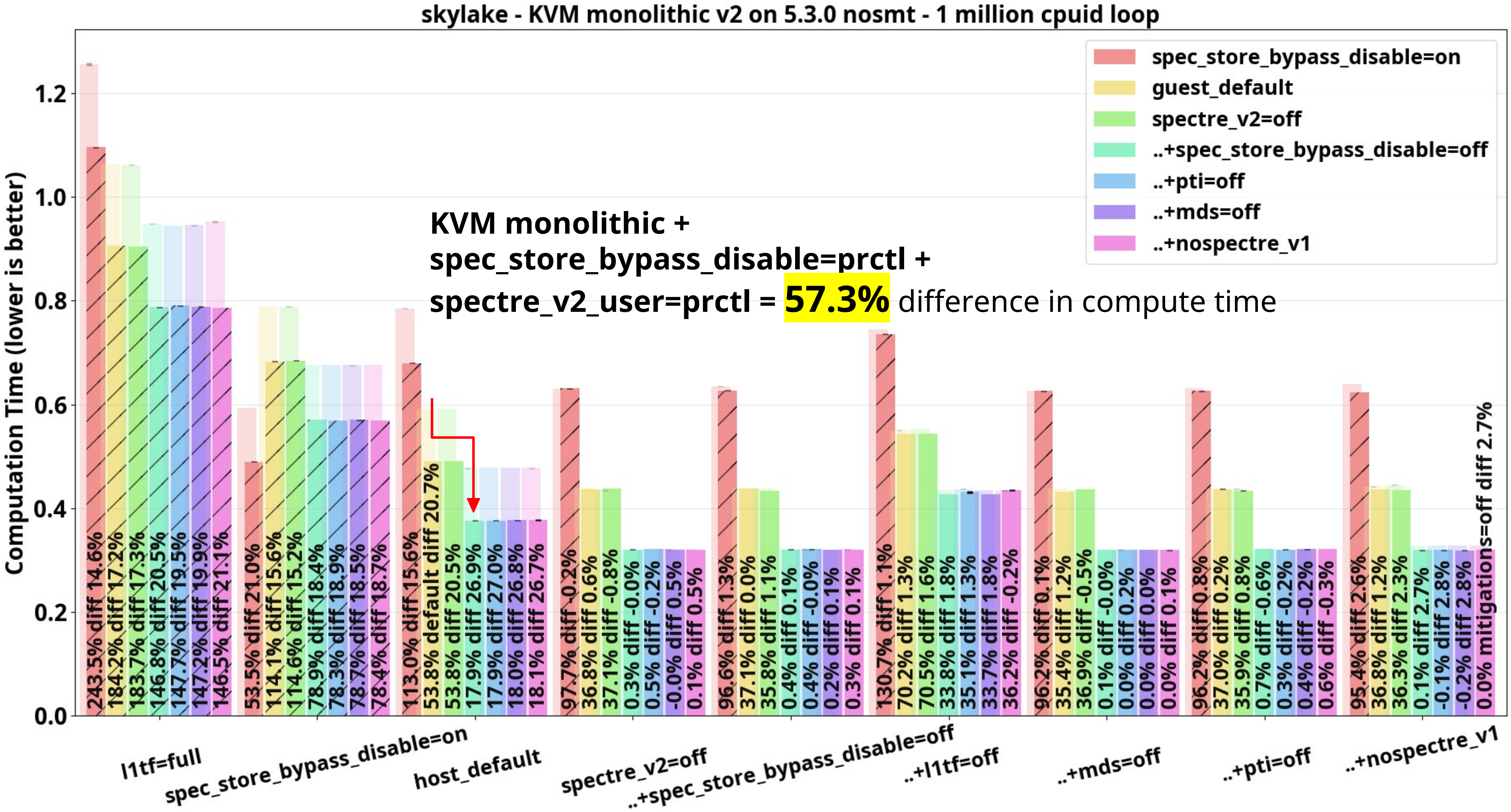

skylake - upstream 5.3.0 KVM nosmt - 1 million cpuid loop
Computation Time (lower is better)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
spec_store_bypass_disable=on
guest_default
spectre_v2=off
..+spec_store_bypass_disable=off
..+pti=off
..+mds=off
..+nospectre_v1
283.5%
224.4%
224.0%
189.5%
188.4%
188.7%
190.7%
80.8%
140.9%
140.7%
106.3%
106.4%
106.3%
106.2%
139.8%
80.7% default
80.5%
45.7%
45.8%
45.7%
45.7%
92.1%
34.0%
32.5%
-2.4%
-2.4%
-2.2%
-2.0%
93.9%
33.6%
33.6%
-2.1%
-2.2%
-2.3%
-2.2%
127.2%
68.0%
68.7%
32.6%
33.3%
32.6%
32.4%
91.1%
33.5%
32.6%
-2.6%
-2.3%
-2.5%
-2.5%
92.6%
33.6%
33.3%
-2.5%
-2.5%
-2.5%
-2.4%
95.2%
34.8%
35.8%
0.1%
-0.0%
-0.1%
0.0% mitigations=off
l1tf=full
spec_store_bypass_disable=on
host_default
spectre_v2=off
..+spec_store_bypass_disable=off
..+l1tf=off
..+mds=off
..+pti=off
..+nospectre_v1



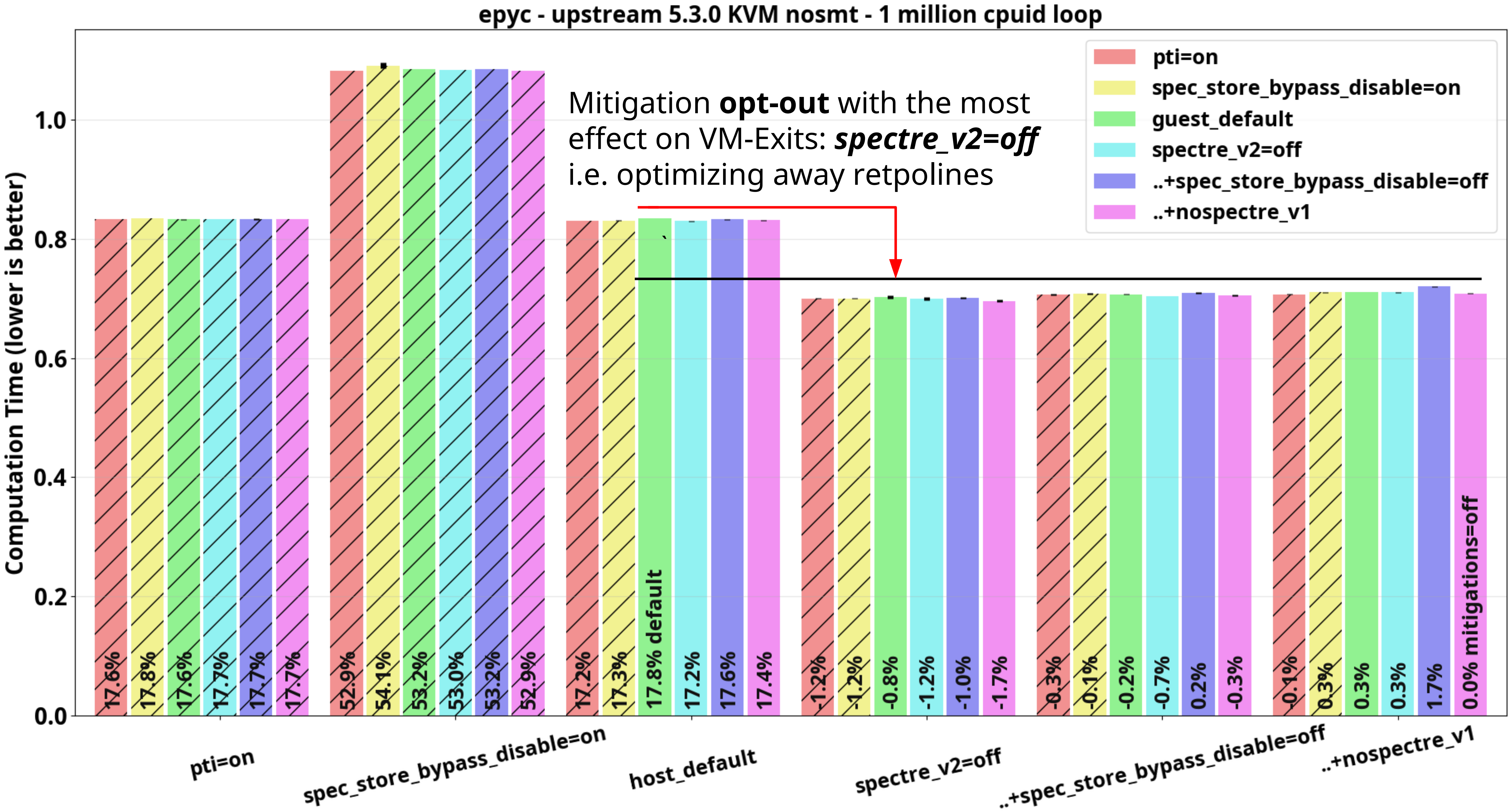
KVM monolithic

20.7% difference in compute time

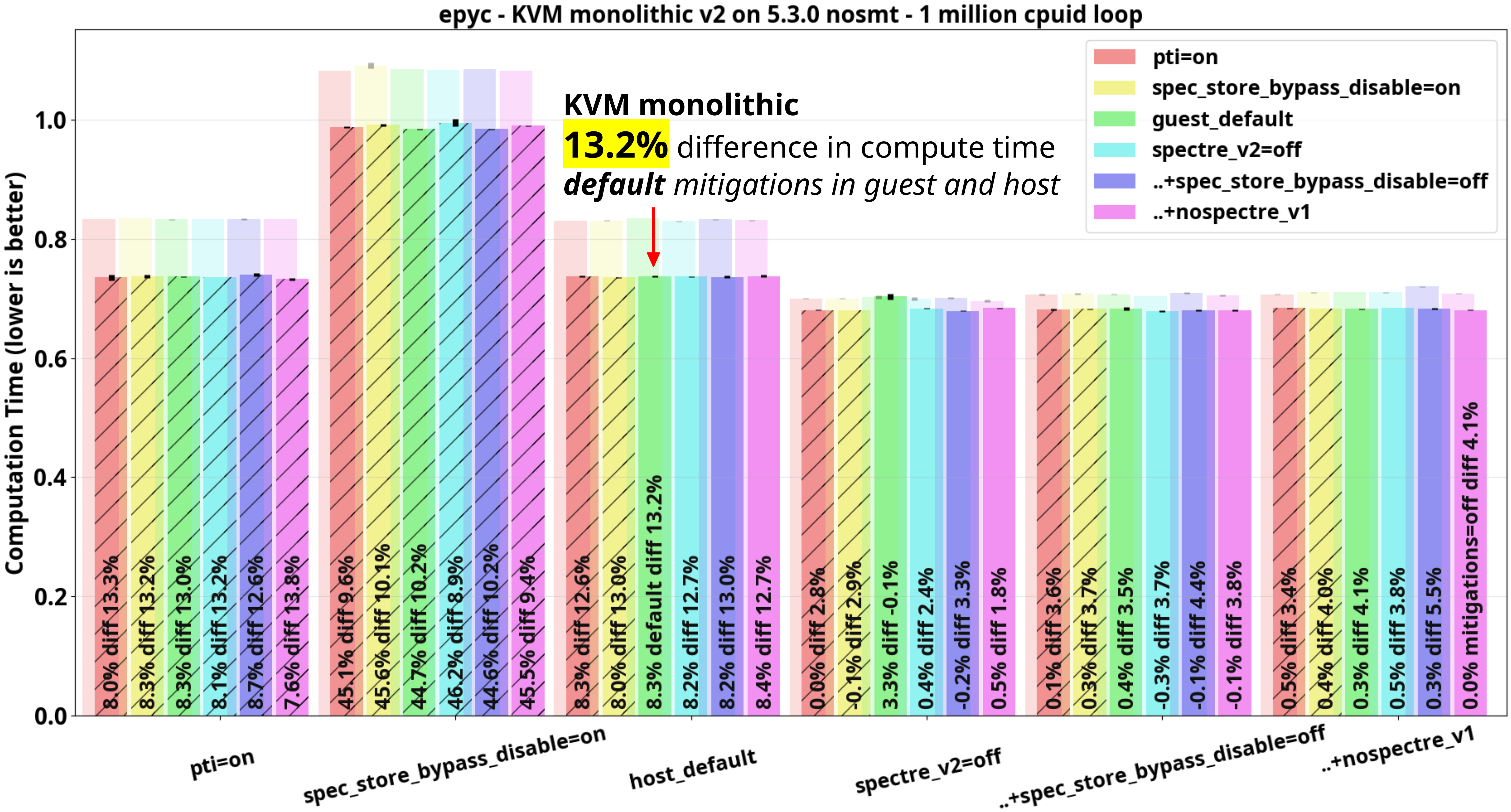
default *mitigations in guest and host*



KVM monolithic +
spec_store_bypass_disable=prctl +
spectre_v2_user=prctl = 57.3% difference in compute time



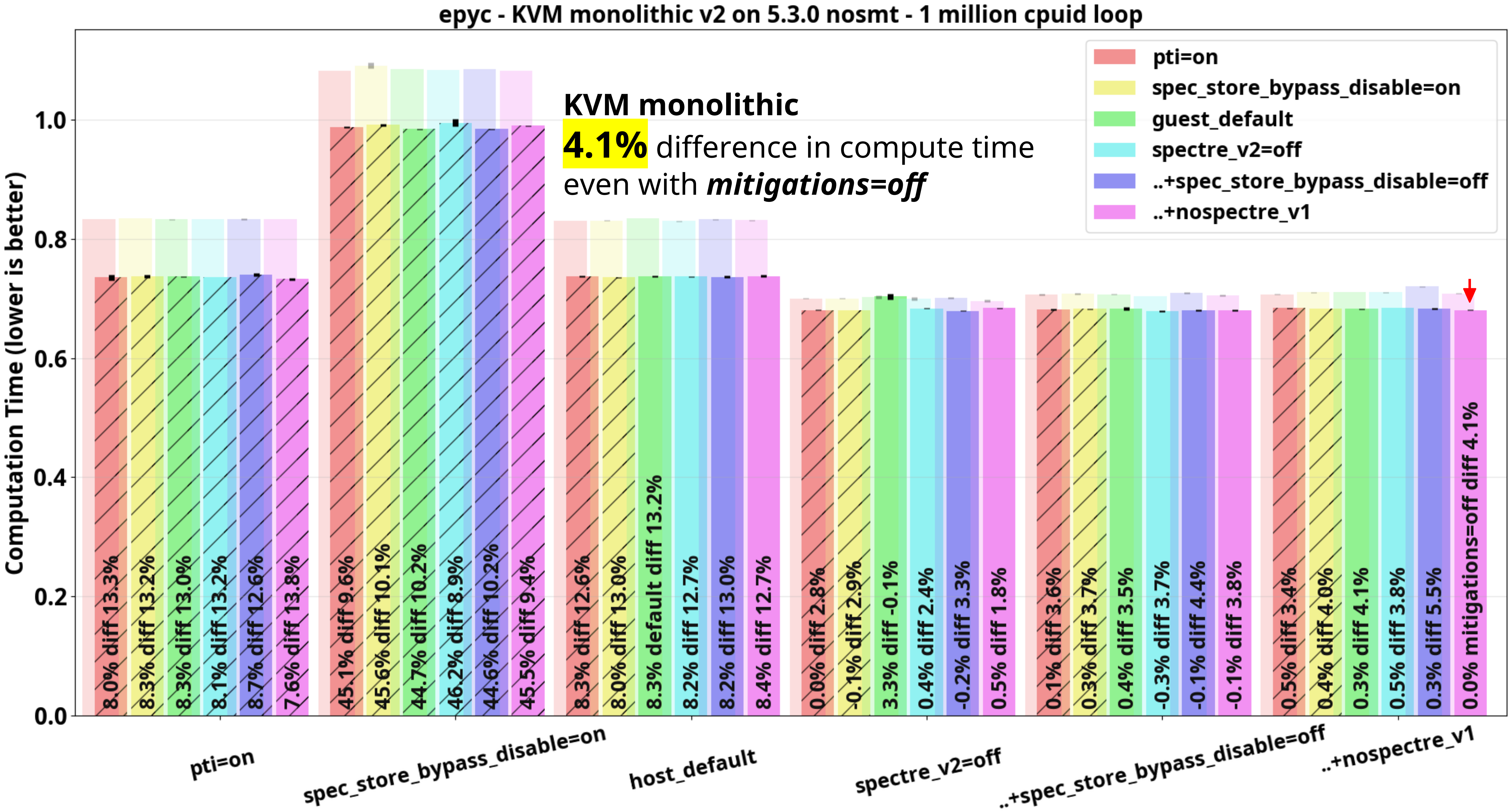
Mitigation **opt-out** with the most
effect on VM-Exits: ***spectre_v2=off***
i.e. optimizing away retpolines



KVM monolithic

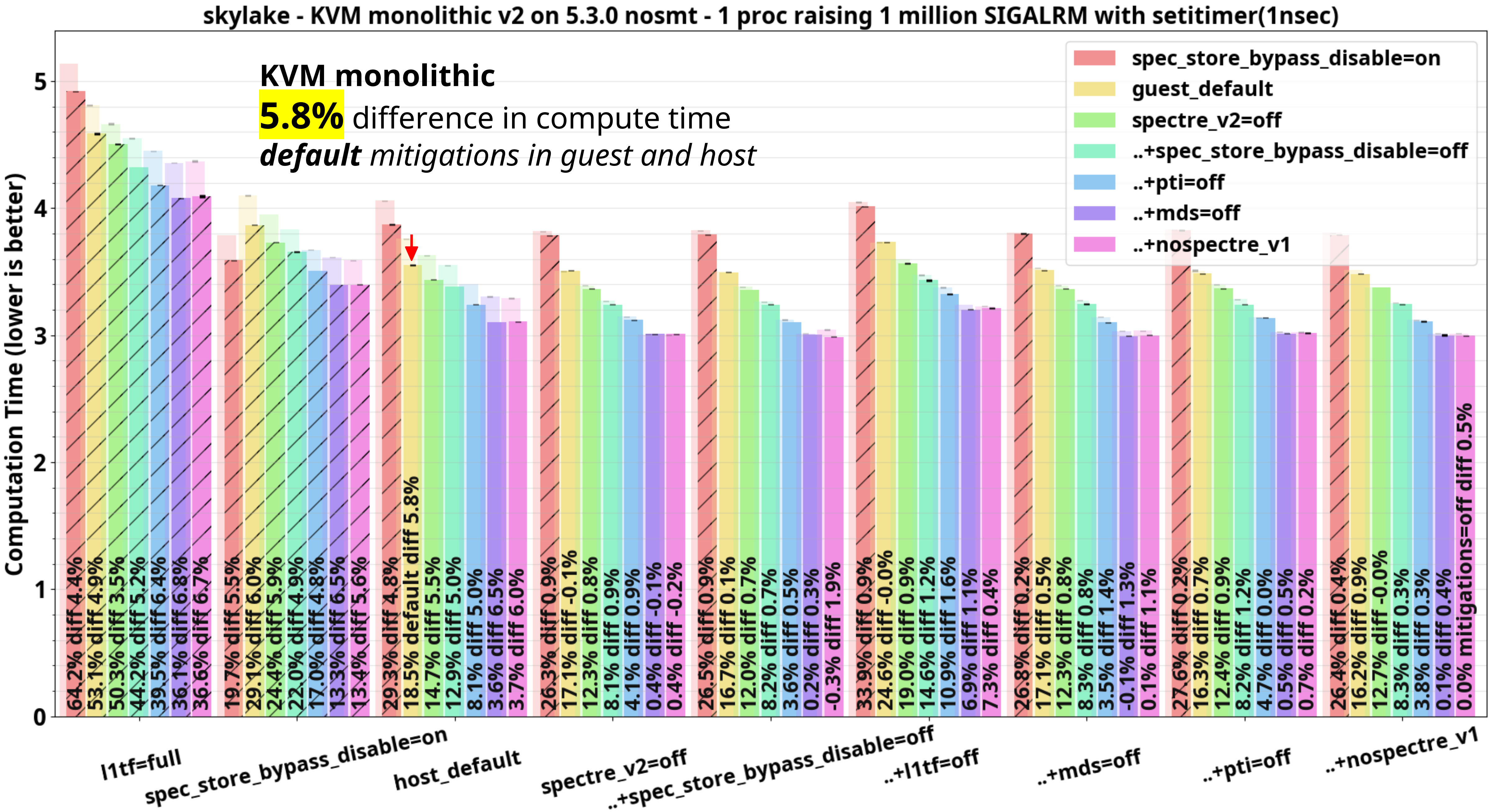
13.2% difference in compute time

default *mitigations in guest and host*



KVM monolithic

4.1% difference in compute time
even with ***mitigations=off***

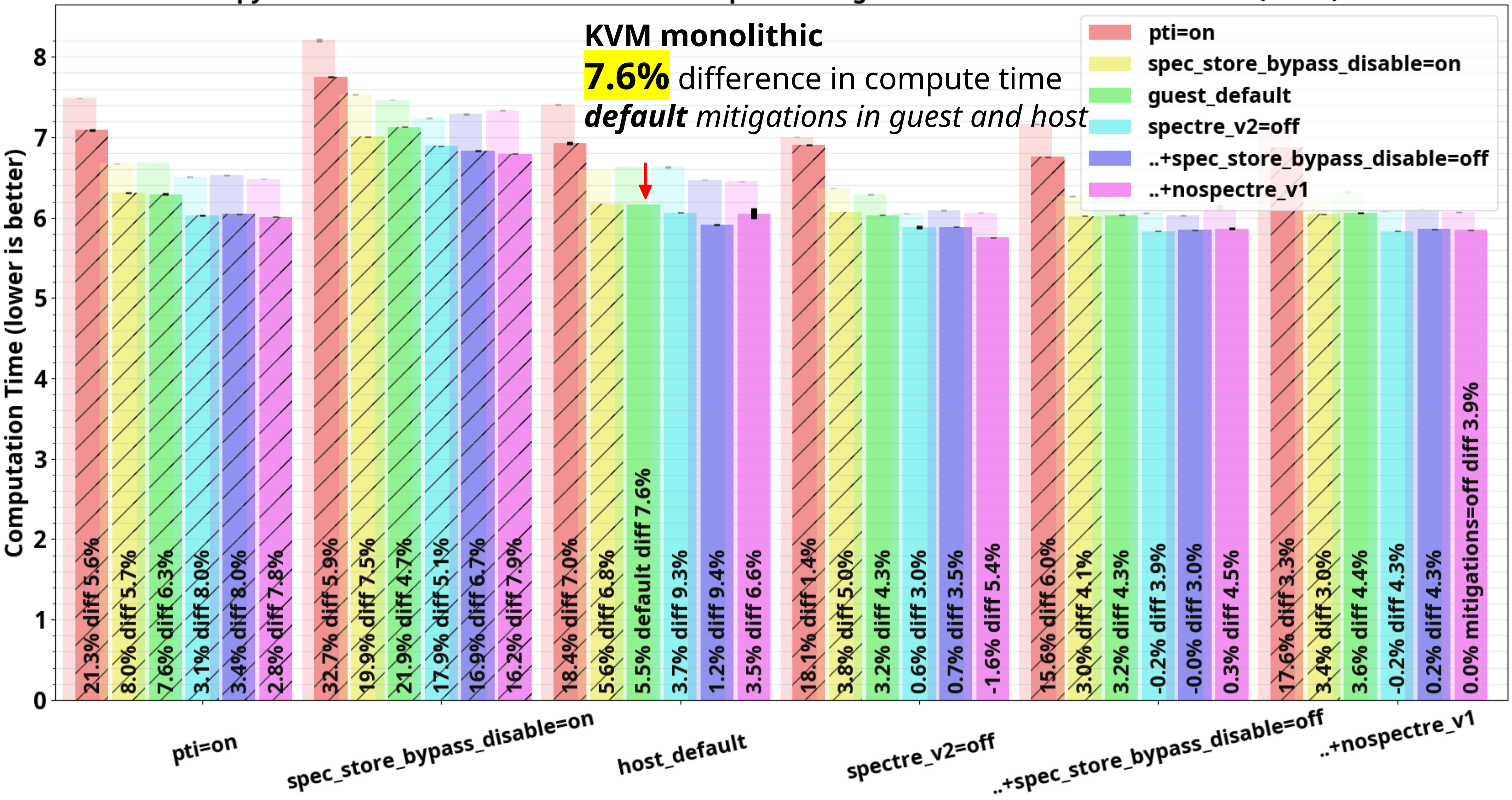


KVM monolithic

5.8% difference in compute time

default *mitigations in guest and host*

epyc - KVM monolithic v2 on 5.3.0 nosmt - 1 proc raising 1 million SIGALRM with setitimer(1nsec)



KVM monolithic

7.6% difference in compute time

default *mitigations in guest and host*

KVM monolithic status

- ***KVM monolithic kernel patch-set posted on kvm@ and lkml@***
  https://lkml.kernel.org/r/20190928172323.14663-1-aarcange@redhat.com
- Kbuild options need more adjustment
- Some warnings from duplicated exports
- Final cleanup of ***kvm_x86_ops*** pending because it can be done incrementally (cleaner)
  - ***kvm_pmu_ops*** already removed

Thank you

Red Hat is the world’s leading provider of

enterprise open source software solutions.

Award-winning support, training, and

consulting services make

Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

youtube.com/user/
RedHatVideos

facebook.com/redhatinc

twitter.com/RedHat

Red Hat