Core Scheduling

Status Update

Vineeth Remanan Pillai
Julien Desfossez
Joel Fernandes

A brief history of side-channel attacks (Summary)

- There are no mitigations that are SMT-safe for cpu vulnerabilities like L1TF and MDS
    - Attack by leaking information from shared resources (caches, micro-architectural buffers) of a core
    - Mitigations mostly involve cache flush and micro-architectural buffer flushes on privilege boundarie switches, but concurrent execution on siblings cannot leverage this.
- So the current state is:
    - Process running on a logical CPU cannot trust the process running on its sibling
    - Disabling SMT is the only safe option
- Disabling SMT has a noticeable performance impact on several types of workloads
- What if, we can make sure that non-trusting threads never gets to share resources exposed by SMT?

Core Scheduling: Concepts

- Have a core wide knowledge when deciding what to schedule on cpu instances
- Grouping of trusted tasks and a mechanism to quickly search for a runnable trusted task in a group
- Forcing a sibling to not run any tasks if it cannot find a runnable trusted task in the same group as the other sibling
- Load balance the cpus so that groups of trusted tasks are spread evenly on the siblings.
    - When a cpu is forced idle, search for a runnable task with matching cookie and migrate it to the forced idle cpu.

Core Scheduling Concepts

- Core wide knowledge used when scheduling on siblings
    - One sibling's rq is selected to store the shared data and that rq->lock becomes the core wide lock for core scheduling.
- While picking the next task in __schedule() if a tagged process is selected, we initiate a selection process
    - Tries to pick the highest priority task from all the siblings of a core and then matches it with a trusted task from the other sibling.
        - If the highest priority process is tagged, find a process with same tag on the other sibling
        - If the highest priority process is untagged, highest untagged process from the other sibling is selected.
        - If a match cannot be found on a sibling, it is forced idle

Core Scheduling Implementation details

- Grouping trusted processes together
  - cpu cgroups: processes under a cgroups are tagged if cpu.tag = 1
  - Cookie is a 64bit value - using the task group address
  - Quick and easy implementation for the initial prototype - Not final
- Tracking tagged processes
  - rq maintains an rbtree ordered by cookie
  - Only tagged processes enqueued
  - Allows to quickly search for a task with a specified tag when trying to match with a tagged task on the sibling.

Core Scheduling: Iterations

- Initial implementation (v1)
    - https://lwn.net/Articles/780084/
- v2
    - https://lwn.net/Articles/786553/
    - Build and stability fixes
- v3
    - https://lwn.net/Articles/789756/
    - Bug fixes and performance optimizations
- v4(5.3.5)
    - https://lwn.net/Articles/803652/
    - Core wide min_vruntime
- v5(5.5.5)
    - https://lwn.net/Articles/813808/
    - Load balancing improvements
    - Stability fixes
- v6(5.7.6)
    - https://lwn.net/Articles/824918/
    - Kernel protection during interrupts
    - Cleanup and fixes aimed for upstreaming

V6 Status

- Round of code cleanup in preparation for upstream
- Documentation done in preparation for upstream
- Performance better than nosmt for most production workloads
  - Synthetic tests like uperf shows considerable performance degradation
- Gaps
  - Cross cpu vruntime comparison is not perfect and may break fairness
  - Coresched awareness logic in load balancing is not perfect and may break fairness
  - Hot plug related crashes present in V6. Fix is posted in v6 discussion
  - Protection for IRQs patch is not perfect - Fix posted in v6 discussion
    - New patch generalizes IRQ protection to kernel protection
  - No consensus on User interface/API yet.

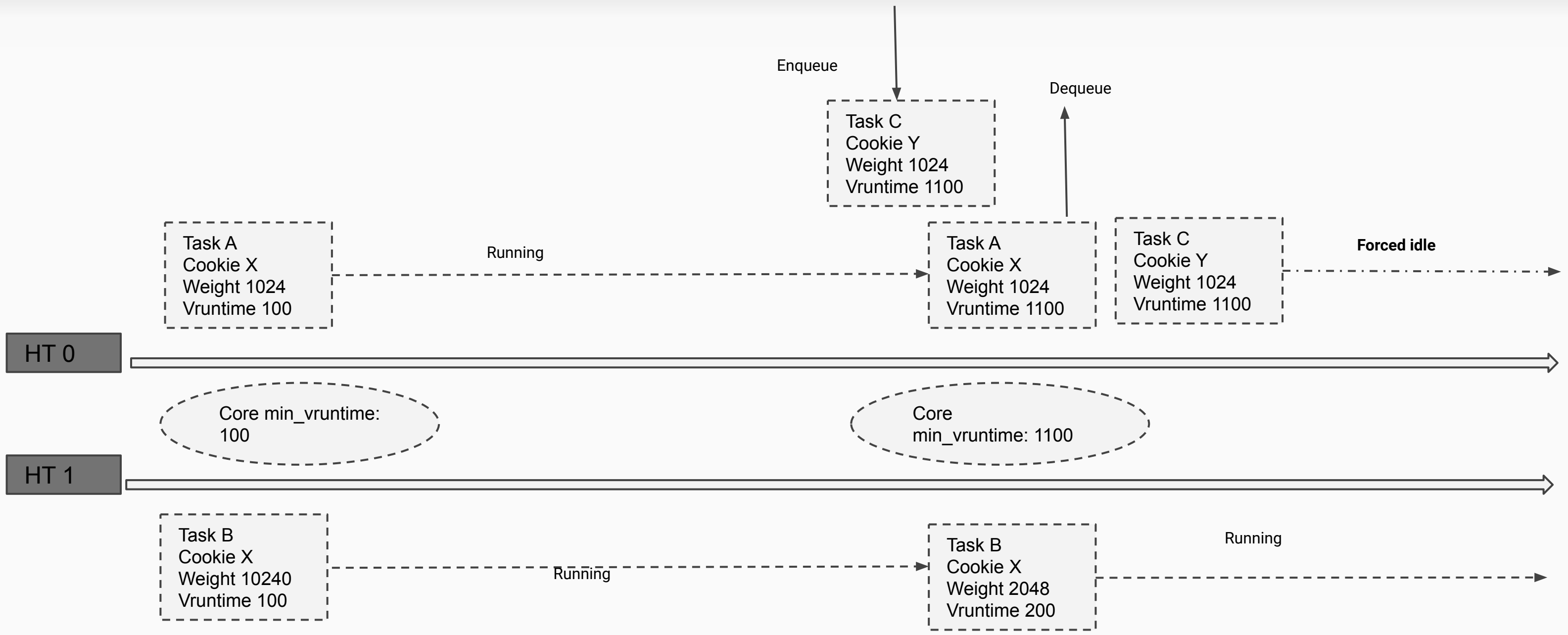
Vruntime comparison across cpus

- Problem
  - We need to compare tasks vruntime across cpus to decide on the high priority task in the core
  - Task vruntime is per cpu and not comparing vruntime across cpu does not make sense.
- Current Solution
  - Core wide min_vruntime: min_vruntime of one sibling is chosen to be the core wide min_vruntime
  - To compare vruntime of tasks in 2 cpus, use vruntime of the parent that is direct child of root cfs_rq(rq->cfs)

Core wide min_vruntime: Issues

- Breaks fairness
  - If tasks with same tag, but different weights - say tasks t1(lower weight) and t2(higher weight) are executing on siblings, lt1's minvruntime increases at a faster rate and is used to set core wide min_vruntime
  - If a new task t3 with a different tag than t1 and t2 gets enqueued on the cpu where t1 is running, it will get starved by t2 as t2's vruntime is very low and t3 starts with t1's vruntime(core wide min_vruntime)

Core wide min_vruntime: Fairness issue



Vruntime comparison: proposed solution

- https://lwn.net/ml/linux-kernel/20200506143506.GH5298@hirez.programming.kicks-ass.net/
- When a sibling is forced idle, we could safely assume it to be a single runqueue(for SMT2)
- So, we sync the vruntime when a sibling goes out of forced idle. The runqueues maintain independant min_vruntime for rest of its time and sync happens only when a sibling comes out of forced idle.
- Downside is - how to extend this notion to SMTx?

Load balancing and task migration

- Load balancing and task migration are not core scheduling aware
  - If a cookie is running tagged task and if untagged tasks are migrated to it, then there might be task starvation

Core Scheduling: v6+ changes

- Hotplug fixes
- Kernel protection from siblings

Core Scheduling: Hot plug issues

- Pick_next_task needs to iterate over all siblings
  - Uses smt_mask
- Schedule can happen during a hotplug event
  - Current cpu might not be there in smt_mask when schedule gets called to switch to idle during offline
  - Sibling might be removed from smt_mask during pick_next_task
- In v5, cpu_offline() was used to bail out, but this was not adequate and had corner cases.
- current cpu should be considered in pick_next_task even if not in smt_mask
  - Schedule idle on offline and migration thread on online
- Retry logic if change in smt_mask is detected in the middle of pick_next_task

Kernel protection

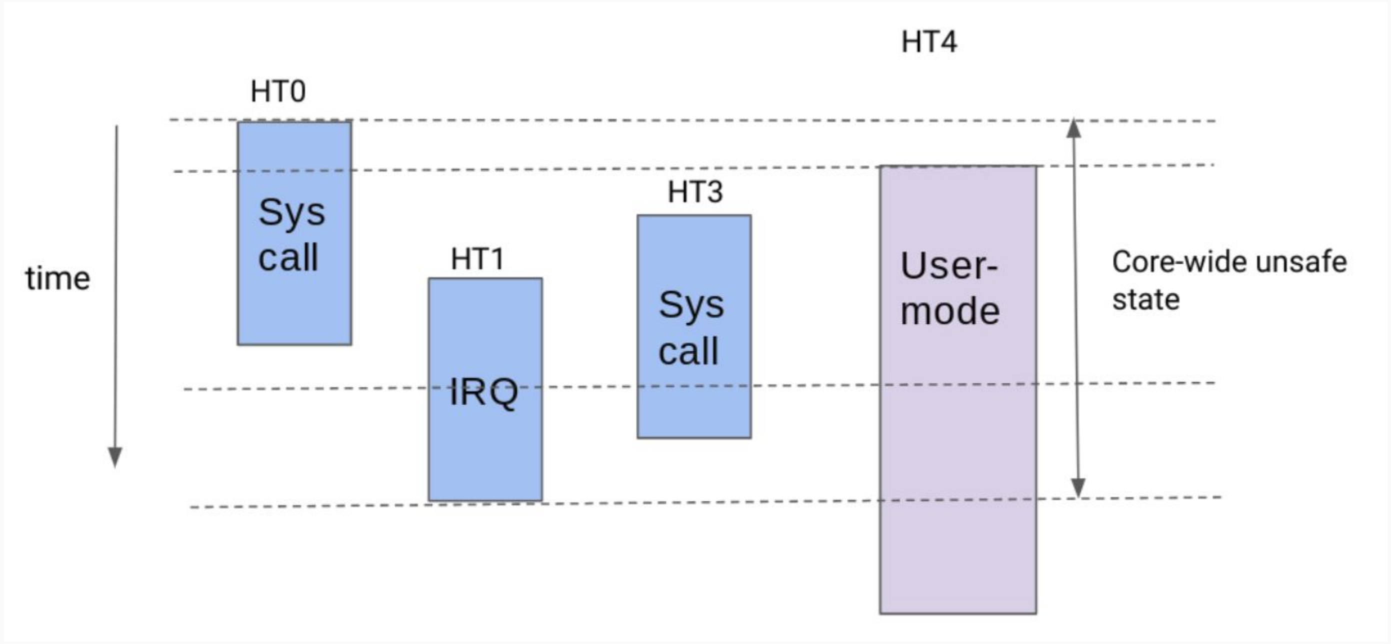
Problem:

Across an ht within a core:

- Core-sched protects user mode victim from user mode attacker.
- No protection of kernel mode victim from user mode attacker.

Kernel protection

Solution : Add a notion of core-wide unsafe state by counting.



Kernel protection

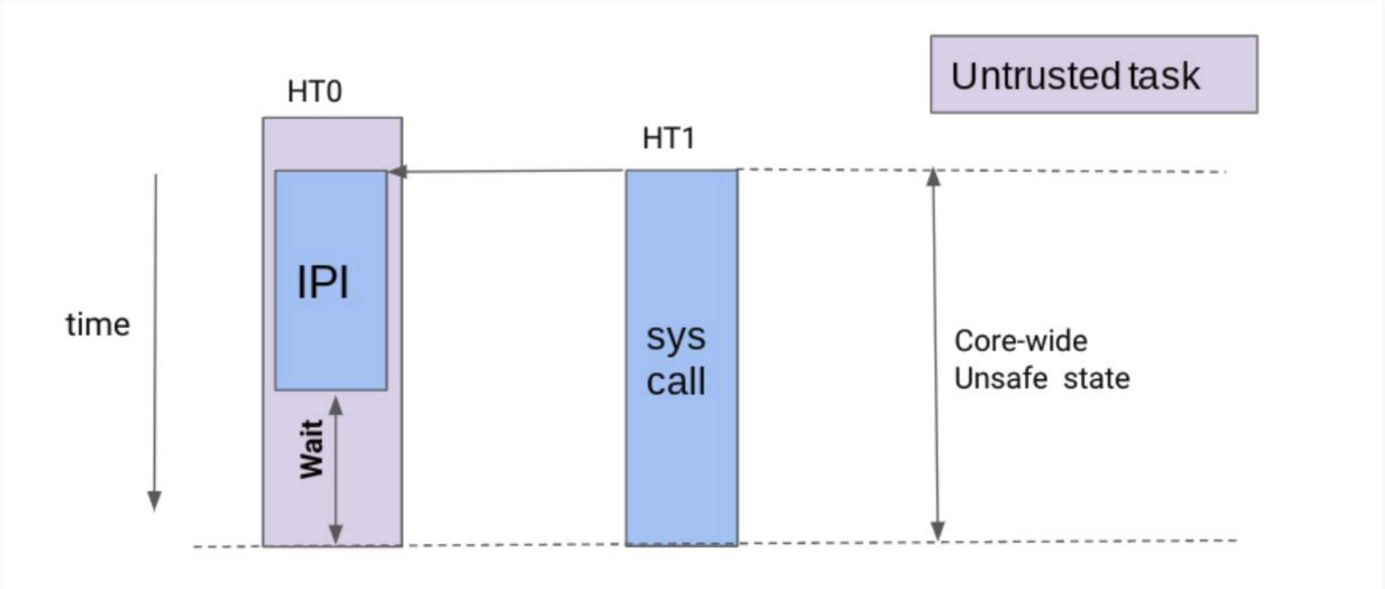
Solution : All counting of kernel entry and exit is done by tracking:

- Usermode entry and exit
- Idle entry and exit.
- VM entry and exit.

This mechanism handles all corner cases and is stable.

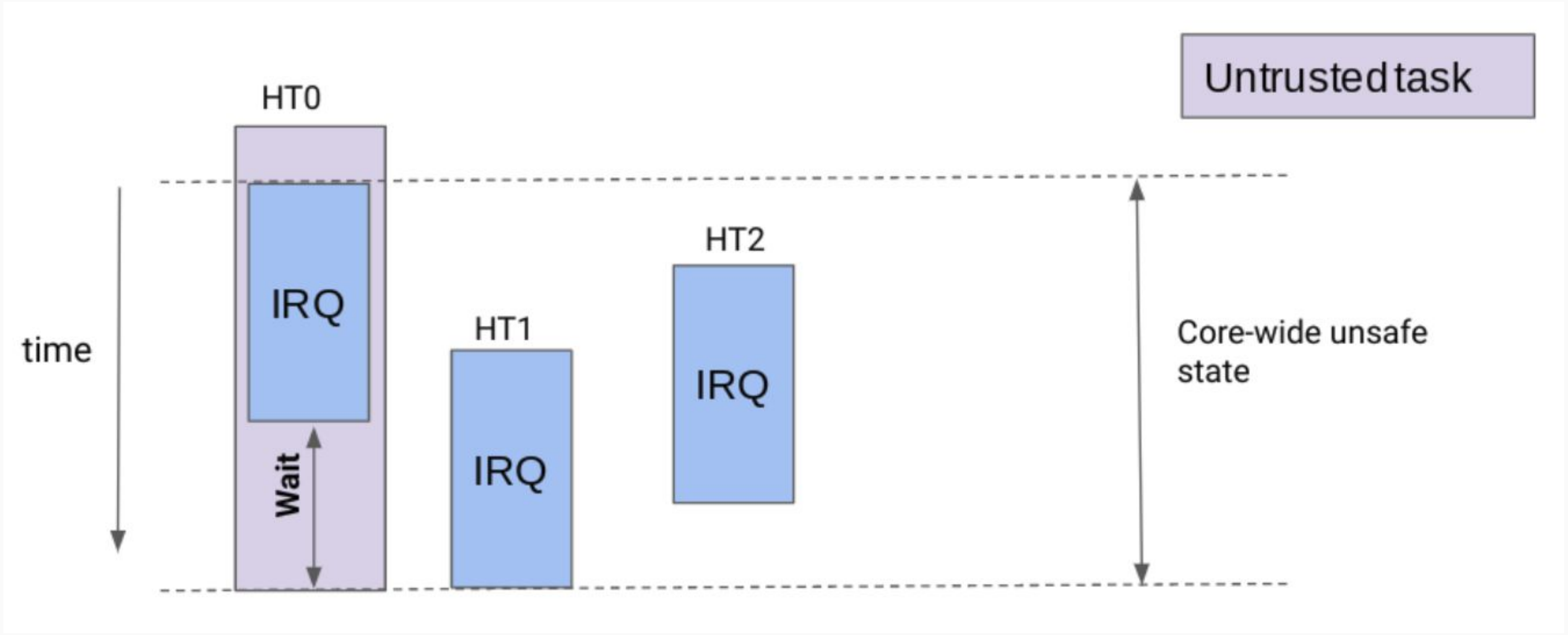
Kernel protection

Solution : Send IPI on outermost core-wide entry if sibling runs untrusted task.



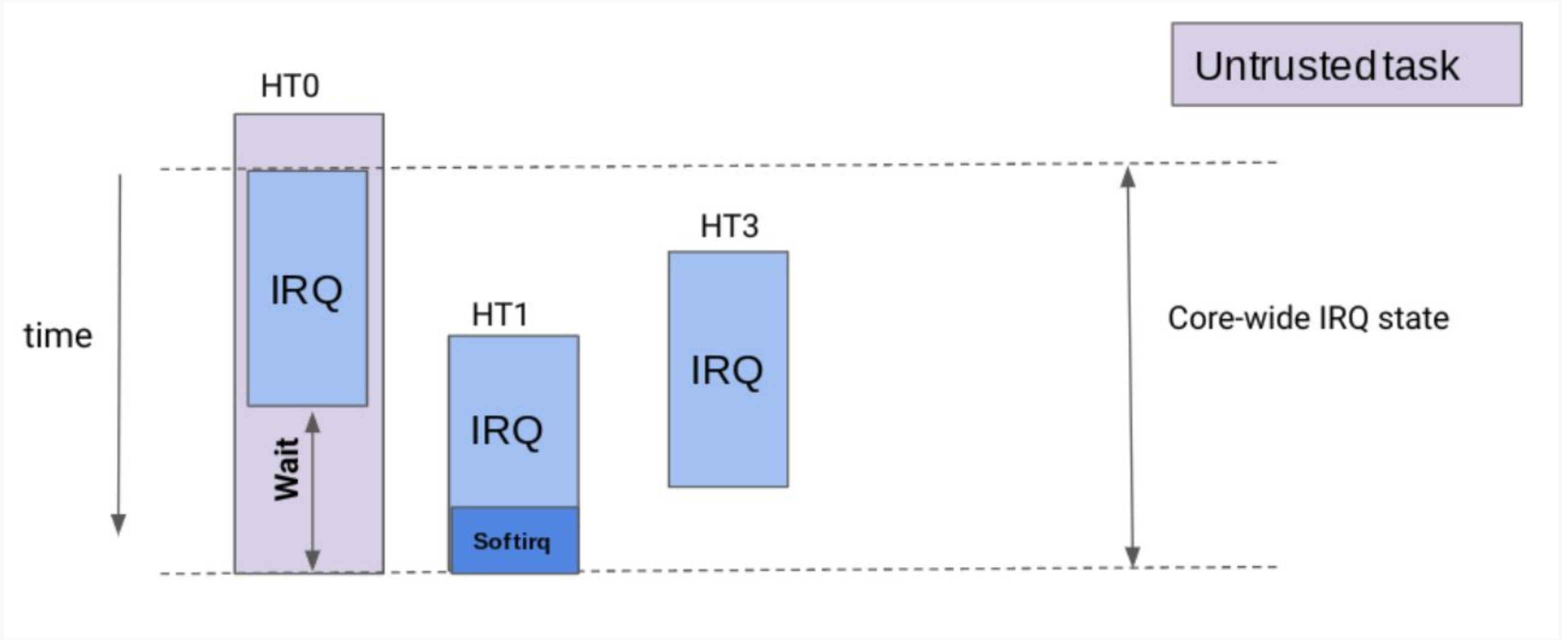
Kernel protection

Solution : Note that an IPI need not always be sent.



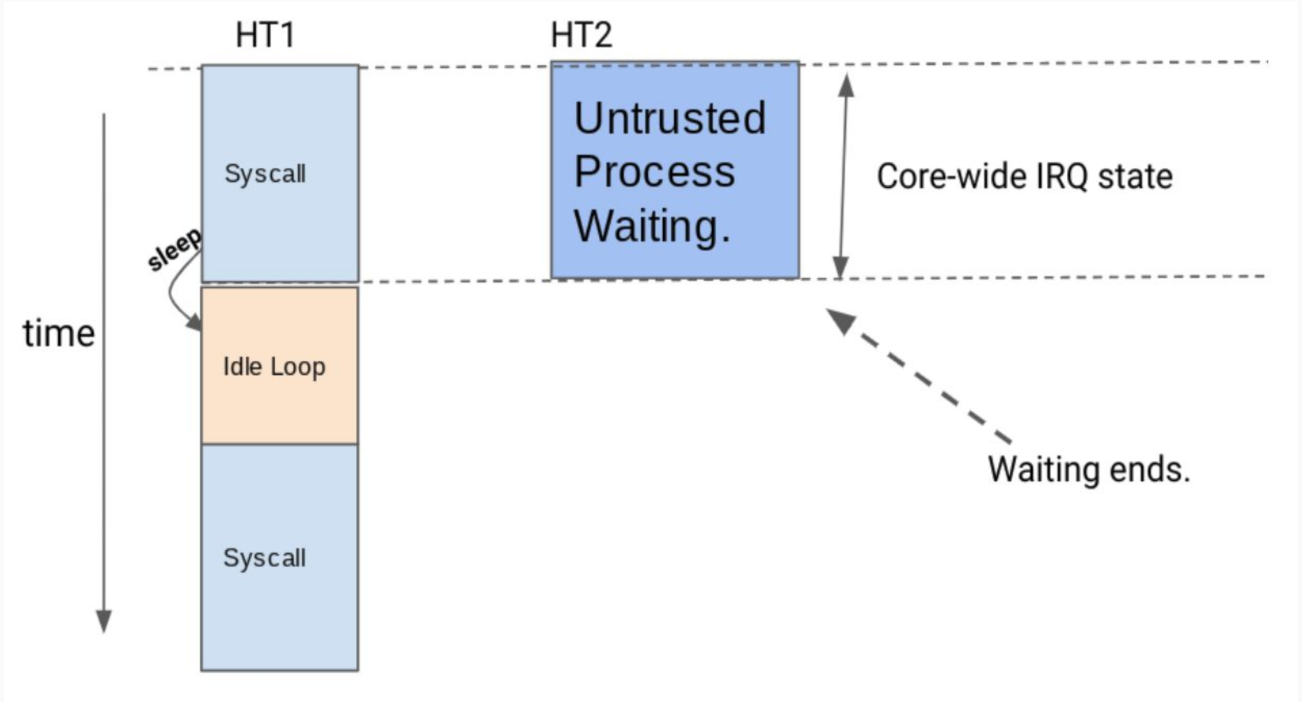
Kernel protection

Solution : Softirqs are also protected by the technique!



Kernel protection

Solution : Switching into idle ensures that we don't wait longer!



Kernel protection

Patch series to do this is posted to LKML:

https://lwn.net/Articles/828889/

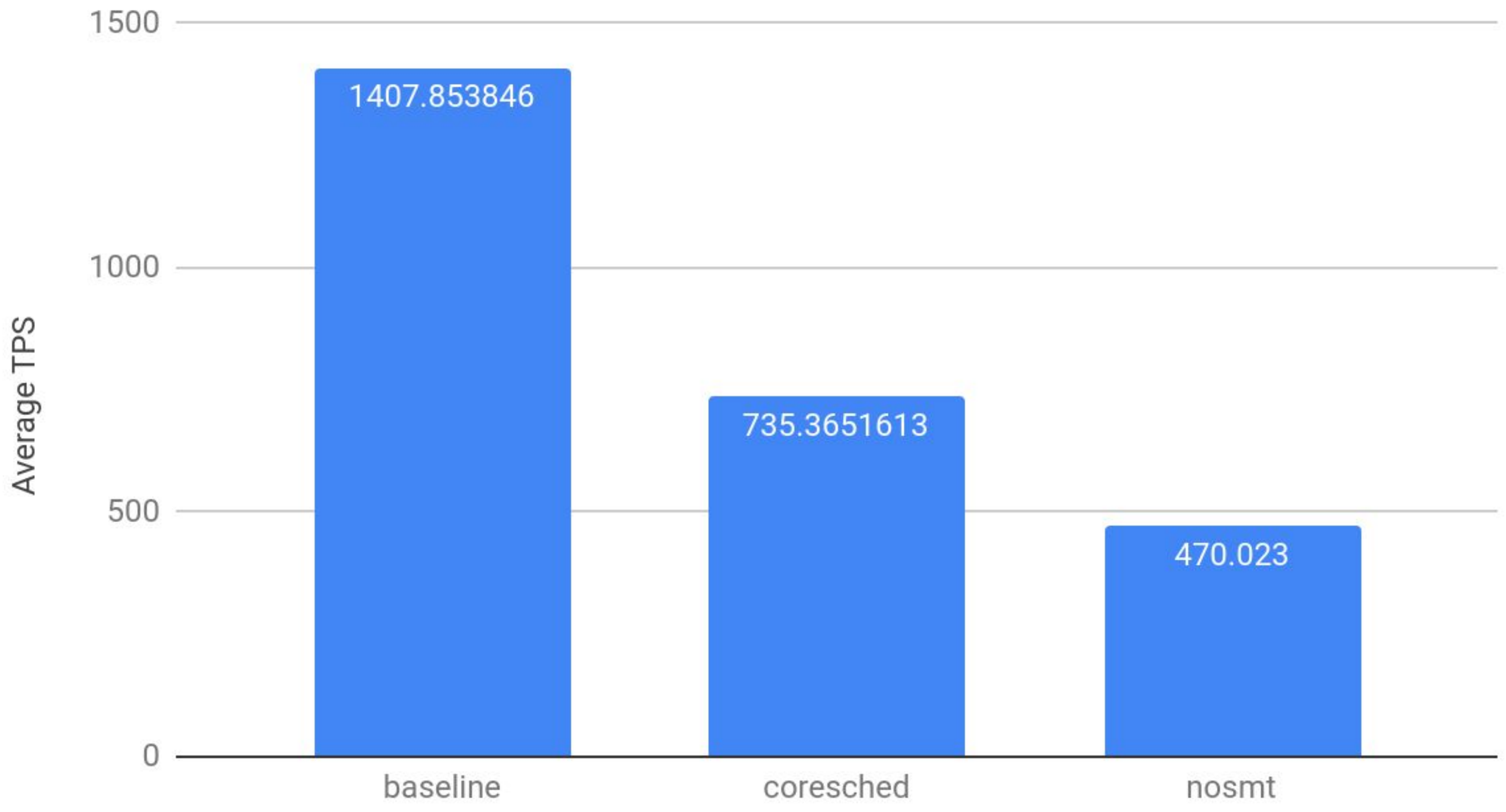
VM-based performance results

- Test environment
  - 2x18 cores (72 hardware threads), Skylake
  - KVM-based VMs
  - V6+protection on v5.8.0-rc6
  - Workloads started with cloudbench (cbtool)
  - Guests running Ubuntu 18.04
- Test descriptions
  - 2x12-vcpus TPCC VMs + 192 noise VMs
    - 1 database VM pinned on each NUMA node
    - 96 semi-idle VMs floating on each NUMA node
    - Mix of CPU, network and disk activity
    - Client VMs running on another physical host, 48 client threads each
    - 3:1 vcpus to hardware threads (becomes 6:1 with nosmt)

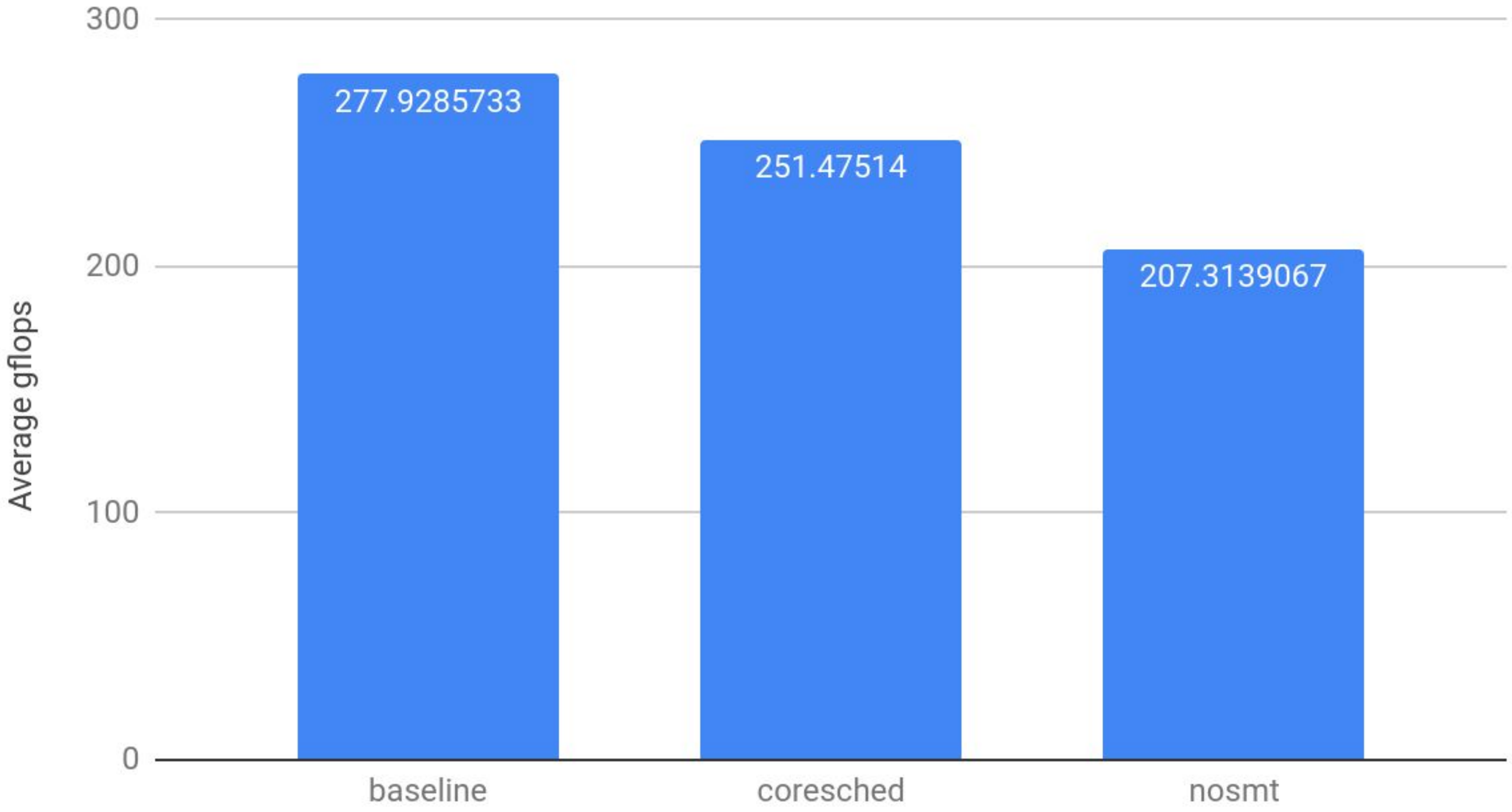
VM-based performance results

- Test descriptions (cont)
  - 3x12-vcpus linpack VMs on 1 NUMA node
    - Pure CPU workload
    - 1:1 vcpus to hardware threads (becomes 2:1 with nosmt)
  - 6x12-vcpus linpack VMs on 1 NUMA node
    - Same, but 2:1 becomes 4:1 with nosmt
  - 4x4-vcpus netperf VMs on 1 NUMA node
    - Pure network, 2gbps cap
    - no difference between the various test cases, not plotted here

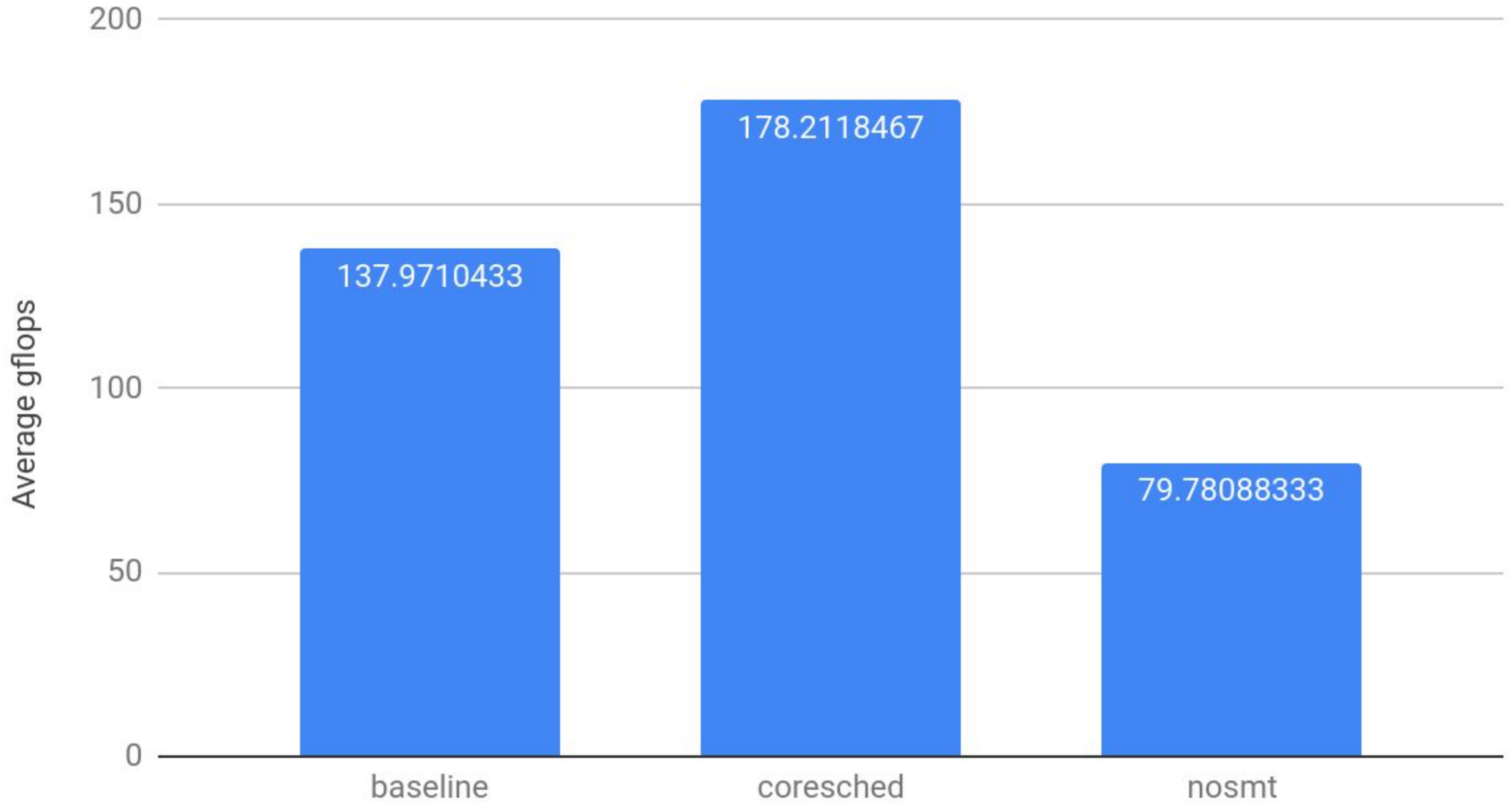
2x12-vcpus TPCC + 192 noise VMs



3x12-vcpus linpack on 1 NUMA node (1:1 --> 2:1)



6x12-vcpus linpack on 1 NUMA node (2:1 --> 4:1)



Test results summary

2x 12-vcpus tpcc + 192 noise VMs			
	avg	stdev	diff
baseline	1407.853846	419.0562531	
coresched	735.3651613	395.7803232	-47.77%
nosmt	470.023	78.68026336	-66.61%
4x netperf on 1 node			
	avg	stdev	diff
baseline	1785.8515	51.23829163	
coresched	1779.802	52.71392881	-0.34%

6x12-vcpus linpack on 1 node (2:1 --> 4:1)			
	avg	stdev	diff
baseline	137.9710433	12.77051941	
coresched	178.2118467	86.44221023	29.17%
nosmt	79.78088333	1.77304606	-42.18%
3x12-vcpus linpack on 1 node (1:1 --> 2:1)			
	avg	stdev	diff
baseline	277.9285733	7.543372368	
coresched	251.47514	20.38683759	-9.52%
nosmt	207.3139067	11.24581545	-25.41%

Thank You!