Core Scheduling

Taming the hyper-threads to be secure!

Vineeth Pillai <vpillai@digitalocean.com>
Julien Desfossez <jdesfossez@digitalocean.com>
Agenda

- Problem Statement
- Core Scheduling Introduction
- Core Scheduling Implementation
- Testing & benchmarking
- Core Scheduling Future work
- Conclusion
A brief history of side-channel attacks

- **Meltdown**
  - Exploits the out of order execution during an exception
  - Data left in L1 cache after out of order execution effects are reverted
  - Attack during context switch to kernel in the same cpu
  - Fix: Page Table Isolation

- **Spectre variant 1**
  - Exploits the speculative execution of conditional branches
  - Data left in L1 cache after out of speculation execution effects are reverted
  - Attack from userspace to kernel.
  - Process attacks are possible if a process(attacker) can pass data to another(victim)
  - Fix: usercopy/swapgs barriers, __user pointer sanitization
A brief history of side-channel attacks (Contd...)

● Spectre variant 2
  ○ Exploits the speculative execution of indirect branches
    ■ Data left in L1 cache after out of speculation execution effects are reverted
  ○ Attacks possible from userspace to kernel, user process to user process, VM to Host and VM to VM
  ○ Fix: Hardware(IBPB, IBRS, STIBP), Software(retpoline)

● L1TF
  ○ Exploits the speculative execution during a page fault when the present bit is cleared for a PTE
    ■ Data left in L1 cache after out of speculation execution effects are reverted
    ■ Any physical page can be leaked
  ○ Fix: L1D cache flush on privilege boundaries
A brief history of side-channel attacks (Contd…)

- **MDS**
  - Data leak from microarchitectural temporary buffers
- **MSBDS**
  - Store buffer not shared across Hyper-Threads, but repartitioned on entering idle
- **MFBDS**
  - Fill buffer shared
- **MLPDS**
  - Load ports shared
- **MDSUM**: Special case of all the above

**Fix**

- **L1TF Vulnerable CPUs**
  - L1TF mitigations fixes MDS as well
- **Non L1TF CPUs**
  - CPU Buffers flush on privilege boundaries
A brief history of side-channel attacks (Summary)

- There are no mitigations that are SMT-safe for 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 boundary 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: task match

CPU 1 and CPU 2 are siblings
(Run queues show tasks stacked in the increasing order or priority)

Run queue of CPU 1
- Task A (ungrouped)
- Task B: Group X
- Task C: Group Z

Run queue of CPU 2
- Task M: Group X
- Task N: Group Z
- Task O: (ungrouped)

Schedule event

Task C > Task O

If task C has more priority than Task O
Schedule Task C and Task N together (same group)
else
Schedule Task O and task A (no groups)
Core Scheduling: no task match

CPU 1 and CPU 2 are siblings
(Run queues shows tasks stacked in the increasing order or priority)

Run queue of CPU 1
- Task A: (ungrouped)
- Task B: Group X
- Task C: Group Z

Run queue of CPU 2
- Task M: (ungrouped)
- Task N: Group X
- Task O: (ungrouped)

Task C > Task O

Schedule event

If task C has more priority than Task O
- Schedule Task C on CPU1 and force CPU2 to be idle
- (Task C do not have a matching task in CPU2's rq)

else
- Schedule Task O and Task A

(Task C is not ready to be scheduled, singulting)

If task O has lower priority than Task C
- Schedule Task O and Task A
Core Scheduling: History

- Core Scheduling patch for KVM
  - vcpu threads trust only other vcpu threads from the same VM

- Generic Core scheduling iteration
  - Generic solution to the initial core scheduling patches
  - Grouping of trusted processes which could be scheduled together on a core.
  - Policy to determine group of tasks that trust each others
    - Initial prototype uses cpu cgroups
      - Quick and easy to prototype
Core Scheduling: KVM based approach

- VCPU threads of the same VM are tagged with a cookie
- To efficiently search for a runnable thread with the same cookie, cookie ordered rbtree in each cpu’s run queue.
- Per core shared data to track the state(sched_domain_shared)
- When a vcpu thread is runnable, it IPI’s its sibling. Sibling on __schedule() checks if there is a matching vcpu thread and if yes, coschedules both
  - On no match, it picks the idle thread so that sibling does not run an untrusted thread.
- Matching logic took care of the various synchronization points
  - VMEXIT, Interrupts, schedule

https://github.com/pdxChen/gang/commits/sched_1.23-base
Core Scheduling Generic Approach

- 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
Core Scheduling: Implementation Issues

- Vruntime comparison across cpus is not perfect
- Forced idle corner cases
- Starvation of interactive tasks when competing with cpu intensive tasks
- Difference in performance between tagged and untagged process
Core Scheduling: vruntime comparison

- Need to compare process priority across the siblings to perform the selection logic.
  - Not straightforward as each queue maintains its own min vruntime
  - V2 fix: Normalize the vruntime when comparing
    - Decrement rq’s minvruntime from task’s runtime
    - Increment sibling rq’s minvruntime to the above before comparing with a task in the sibling
    - Fixes the initial starvation seen in v1.

```c
/*
 * Normalize the vruntime if tasks are in different cpus.
 */
if (task_cpu(a) != task_cpu(b)) {
    vruntime -= task_cfs_rq(b)->min_vruntime;
    vruntime += task_cfs_rq(a)->min_vruntime;
}
```
Vruntime comparison corner cases after normalization

Task a
Cookie: X
a->runtime == rq_1->min_vruntime

Matching logic:
Task a wins the priority
cpu 2 forced idle because
of cookie mismatch

Task b
Cookie: Y
b->runtime == rq_2->min_vruntime

Schedule event happens on cpu 2
Normalize vruntime for task b
(b->runtime - rq_2->min_vruntime) + rq_1->min_vruntime
= 0 + rq_1->min_vruntime
= 0 + vruntime

Task a wins again as the comparison logic favors the first task if
the vrtimes are equal

if (task_cpu(a) != task_cpu(b)) {
  vruntime = task_cfs_rq(b)->min_vruntime;
  vruntime += task_cfs_rq(a)->min_vruntime;
}
return !(s64)(a->se.vruntime - vruntime) <= 0;
Forced idle corner case example

- Each sibling has only one task each, but with different cookies
- One cpu has to go idle forcing its runnable task to be preempted
- Now, the running task if compute intensive would not hit __schedule unless there is any event that triggers schedule.
- Idle thread also will not wake up as task_tick for idle is nop
- So the idle cpu stays on idle for a considerable period until some schedule event happens on either of the siblings in the core
Proposed Solutions

- **Forced Idle Issue**
  - Accounting of forced idle time to trigger scheduling
  - Instead of using idle thread on cpu, introduce a per cpu task that idles so that scheduler does not confuse idle with forced idle
  - Special checks in idle thread to differentiate between idle and forced idle

- **Vruntime comparison across cpu**
  - Check the vruntime of parent entity going all the way to the root entity of the cfs_rq of cpu.
  - Core wide vruntime
Testing methodology

- Correctness validation with perf/LTTng + Python for parsing the CTF traces
  - “Are there incompatible tasks running at the same time on the core?”
  - “Why is a particular task not running while the whole core is idle?”
- Debugging with ftrace (using `trace_printk`)
  - “Why is one task not getting CPU time at that moment?”
- eBPF for runtime efficiency statistics
  - “How much time a running task is off cpu?”
Process 21687 (qemu-system-x86):
- total runtime: 2758219964112 ns,
- local neighbors (total: 1085548229756 ns, 39.357 % of process runtime):
  - qemu-system-x86 (21687): 972049547 ns
  - CPU 9/KVM (21721): 8720208965 ns
  - CPU 3/KVM (21714): 1594287115 ns
  - CPU 0/KVM (21707): 158177274295 ns

[...]
- idle neighbors (total: 1636163538574 ns, 59.320 % of process runtime):
  - swapper/4 (0): 63532547226 ns
  - swapper/10 (0): 4000441661 ns

[...]
- foreign neighbors (total: 2174790665 ns, 0.079 % of process runtime):
  - qemu-system-x86 (22059): 38360466 ns
  - CPU 4/KVM (22085): 11039429 ns

[...]
- unknown neighbors (total: 15999442846 ns, 0.580 % of process runtime)
- **Micro benchmarks with worst cases:**
  - 2 incompatible cpu-intensive tasks each pinned on a different sibling of the same core
  - Over-committed cores

- **Real-world scenarios:**
  - Large busy virtual machines (ex: TPCC benchmark in a 12 vcpus VM)
  - IO intensive VMs
  - CPU intensive VMs
  - Various configurations:
    - Alone on the NUMA node
    - With other similar on the same NUMA node
    - With noise (mostly-idle) VMs
Early performance results: CPU

3x 12-vcpus linpack on 36 logical CPU NUMA node

- Baseline: 270.104 average gflops
- Nosmt: 209.3180167 average gflops
- Core scheduling: 269.53685 average gflops
Early performance results: CPU

- cpu-intensive workloads in multi-vcpu VMs with all physical CPUs used:
  - Core scheduling performs better than when we disable SMT (N-cpu/2)
- If N/2 is not overcommitted nosmt performs better
  - Side note: there are signs that the load balancer should be SMT-aware to place the tasks more adequately
Early performance results: IO

- For IO-intensive workloads:
  - No major difference between no-SMT and core scheduling
Early performance results: mixed resources

2x 12vcpus MySQL benchmark + 92 1-vcpu noise VMs
Running on a 36 logical CPUs NUMA node

<table>
<thead>
<tr>
<th>Test case</th>
<th>Average TPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td>1976.685</td>
</tr>
<tr>
<td>nosmt</td>
<td>1960.293</td>
</tr>
<tr>
<td>core scheduling</td>
<td>1677.297</td>
</tr>
</tbody>
</table>
Early performance results: mixed resources

- In mixed workloads with noise (TPCC benchmark + semi idle VMs) nosmt is more performant than core scheduling if %idle is ~> 40%
Process selection and process matching logic needs a rework
  - Current implementation does not go beyond the highest priority task in each class.

Syscalls/interrupts and VMEXIT can cause kernel to be co-scheduled along with a untrusted task in the user space and would need protection
  - This might be very costly
  - L1TF or VM-only workloads, needs only protection on VMEXIT
    - This was done in the first iteration of core scheduling (KVM based)

Define the right interface to group trusted processes
  - cgroup is currently used because it was easy for prototyping
Thank You!

- Discussions to continue @ core scheduling MC
  - [https://linuxplumbersconf.org/event/4/contributions/430/](https://linuxplumbersconf.org/event/4/contributions/430/)

- Resources
  - [https://github.com/pdxChen/gang/commits/sched_1.23-base](https://github.com/pdxChen/gang/commits/sched_1.23-base)
  - [https://lwn.net/Articles/780703/](https://lwn.net/Articles/780703/)
  - [https://lwn.net/ml/linux-kernel/20190218165620.383905466@infradead.org/](https://lwn.net/ml/linux-kernel/20190218165620.383905466@infradead.org/)
  - [https://lwn.net/ml/linux-kernel/cover.1556025155.git.vpillai@digitalocean.com/](https://lwn.net/ml/linux-kernel/cover.1556025155.git.vpillai@digitalocean.com/)
  - [https://lwn.net/ml/linux-kernel/cover.1559129225.git.vpillai@digitalocean.com/](https://lwn.net/ml/linux-kernel/cover.1559129225.git.vpillai@digitalocean.com/)