Deep dive in the scheduler

Vincent Guittot
PMWG / Linaro
Introduction

- "The Linux Kernel Scheduler - Overview" session is a good introduction

- Scheduling class

  - Stop: will replace any running task. A running task can't migrate as an example
  - Deadline: real time tasks with period and runtime properties
  - RT: real time tasks with fixed priority
  - CFS: Completely Fair Scheduler
  - Idle: There is nothing to run

- This session will focus on CFS
- But 55min is short to go through all details
- Will look in details some core parts of the scheduler
Agenda

- Topology
- Metrics
- Cgroup
- Task placement
TOPOLOGY
Scheduler topology

- Describe CPUs topology in the system
  - Follow memory/cache layout

- Made of:
  - Sched_domain: a level of scheduling made of groups of CPUs
  - Sched_group: a group of CPUs in sched_domain

- Scheduler flags describes topology:
  - SDASYM_CPU_CAPACITY: Domain members have different CPU capacities
  - SD_SHARE_CPU_CAPACITY: Domain members share CPU capacity
  - SD_SHAREPOWER_DOMAIN: Domain members share power domain
  - SD_SHARE_PKG_RESOURCES: Domain members share CPU pkg resources
Typical topology

- **SMT**
  - 0-1
  - 2-3
  - 4-5
  - 6-7

- **MC**
  - 0-1
  - 2-3
  - 4-5
  - 6-7

- **DIE**
  - 0-7
  - 8-15

- **NUMA1**
  - 0-15

- **NUMA2**
  - 0-XX

**SD_SHARE_PKG/Resources**: sched_domain, sched_group, SD_SHARE_CPUCAPACITY, SD_SHARE_PKG_RESOURCES
Build domain

● Architecture can set its own topology layer
  ○ Default one can be superseded w/ set_sched_topology
  ○ It must provide cpus mask, flags, name for each level

● All levels are built at init

● Rebuild happens for each topology change:
  ○ Capacity update
  ○ CPU hotplug
  ○ Cgroup partitioning

● Then useless levels are removed
  ○ 1 CPU in domain
  ○ Only 1 group
  ○ No new CPUs compared to child
  ○ No useful information compared to child
**big.LITTLE example**

- Example with difference between b.L and dynamIQ

<table>
<thead>
<tr>
<th>MC</th>
<th>L0</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIE</td>
<td>L0, L1, L2, L3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MC</th>
<th>b4</th>
<th>b5</th>
<th>b6</th>
<th>b7</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIE</td>
<td>b4, b5, b6, b7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Legacy big.LITTLE

- LLC is not at the same level

<table>
<thead>
<tr>
<th>MC</th>
<th>L0</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>b4</th>
<th>b5</th>
<th>b6</th>
<th>b7</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIE</td>
<td>L0, L1, L2, L3, b4, b5, b6, b7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### DynamiQ system
Scheduler debug

- Sched-debug output topology on console:
  - Example on hikey960

```
[ 1.883554] CPU3 attaching sched-domain(s):
[ 1.883555]   domain-0: span=0-3 level=MC
[ 1.883557]     groups: 3:{ span=3 cap=461 }, 0:{ span=0 cap=459 }, 1:{ span=1 cap=457 }, 2:{ span=2 cap=460 }
[ 1.883565]   domain-1: span=0-7 level=DIE
[ 1.883567]     groups: 0:{ span=0-3 cap=1837 }, 4:{ span=4-7 cap=4082 }
[ 1.883572] CPU4 attaching sched-domain(s):
[ 1.883573]   domain-0: span=4-7 level=MC
[ 1.883576]     groups: 4:{ span=4 cap=1020 }, 5:{ span=5 cap=1021 }, 6:{ span=6 cap=1023 }, 7:{ span=7 cap=1018 }
[ 1.883584]   domain-1: span=0-7 level=DIE
[ 1.883586]     groups: 4:{ span=4-7 cap=4082 }, 0:{ span=0-3 cap=1837 }
```

- Note the capacity difference
  - Reflect uarch... but not only
  - It's about current capacity and not absolute capacity (will see that later)
Scalability and performance

- Scalability and performance are key for large system

- Per CPU sched_domain
  - Scalability: minimize contention when accessing the structure

- But shared sched_group
  - With shared value modified at runtime

- Scheduler caches some sensitive levels like:
  - Last level of Cache (aka LLC): point to largest level where groups shares cache
  - Asymmetric CPU level: point to level where groups have CPUs with different capacity
  - See differences in previous example
METRICS
Scheduler Metrics

- Ensure a fair split of CPUs runtime between threads and/or groups of threads
- Adjust CPU performance
- Select the best CPU for a thread

Main metrics:
- vruntime: virtual running time (only for cfs)
- CPU capacity: reflect the compute capacity
- PELT: Per Entity Load Tracking
vruntime
vruntime

- Virtual running time of CFS task
  - Weight the real running time with priority of the task
  - Nice 0 is the reference: vruntime == real runtime
  - Nice < 0: vruntime increases slower than real time
  - Nice > 0: vruntime increases faster than real time

- Provide more running time to task with higher priority (lower nice)
  - Scheduler selects next task w/ lowest vruntime
weight

- Priority -1 gives ~+10% of runtime
  - Precomputed values in sched_prio_to_weight array

- Weight of nice 0 is 1024 / Weight of nice 1 is 1277
  - Why is the ratio ~1.25 ?

<table>
<thead>
<tr>
<th>100% runtime</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>55%</td>
<td>45%</td>
</tr>
</tbody>
</table>

- 10% of difference: $\frac{1277}{(1277+1024)} = 55.5\%$
Running tasks

- Fairness at macroscopic level
  - nice 0 vs nice 1
  - nice 10 vs nice 11
  - nice 18 vs nice 19
sched_period / sched_slice

- **sched_period**: typical time window to run all tasks
  - $6ms \times (1 + \log_2(ncpus)) = 24ms$ on hikey

- **sched_slice**: typical time allocated for a given task
  - 12ms when 2 tasks run on same CPU

- Compared against vruntime
  - vruntime increase faster for higher nice priority (lower priority)

- Other event can impact the sched_period and sched_slice
  - Like a task wake up
Sleep and Migration

- vruntime is local to the CPU on which task runs

- Normalize when task sleep
  - Don’t provide too much runtime to long sleeping task

- Normalize when task migrate
  - Don’t provide too much runtime to a newly migrated task

- Example:
  - TA nice 0 runs 50ms every 200ms
  - TB nice 0 runs 500ms every 2 seconds
  - w/o normalization:
    - TA’s vruntime increase during the 1.5 seconds
    - Then TB might run ~400ms in 1 shot
vruntime at wakeup

- New task starts after current sched_slice

- Wake up task can’t get more than a sched_slice deficit
  - Halve their deficit

- Wake up task preempts current task
  - Ensure minimum runtime before preemting
CPU capacity
CPU capacity

- Reflect the compute capacity of a CPU
  - Default is \texttt{SCHED\_CAPACITY\_SCALE} = 1024
  - Can be superseded by arch w/ \texttt{arch\_scale\_cpu\_capacity}

- Which platform don’t use default capacity?

- Hyperthreading
  - 2 CPUs != 2 cores
  - 1 core w/ HT > 1 core w/o HT

- big.LITTLE
  - Make difference between big and LITTLE cores
CPU capacity

● Compare compute capacity of 2 groups of CPUs
  ○ Evenly spread load on system
  ○ Check that local CPU has enough capacity

● Original capacity:
  ○ max compute capacity of the CPU

● Current capacity:
  ○ Remaining capacity available for CFS
  ○ Remove capacity used by interruption, deadline, RT
  ○ Periodically update to scale
Per Entity Load Tracking
Per Entity Load Tracking

- Track the estimated
  - Utilization
  - Load
  - Runnable load

- Track for RQs and tasks
  - All RQs: deadline, RT, CFS
  - CFS tasks
  - And also interrupt

- Geometrics series
  - Half life period is 32ms
  - 30% in 17ms
  - 60% in 45ms
  - 90% in 105ms
  - 100% in ~340ms
Propagate task migration

- Migrate PELT signal with CFS task
Util_est

- Keep track of last max utilization
  - Minimize frequency toggling
Scale invariance

- Invariant with CPU micro-architecture
- Invariant with CPU frequency
Invariance in deadline

- runtime accounting is scaled
  - with CPU micro-architecture
  - with CPU frequency

- Deadline bandwidth used to set frequency
Scheduler cgroup

- Group some tasks to apply same properties
  - Share nice priority
  - CPU affinity
  - CFS bandwidth

- Hierarchical topology
  - Childs are subset of parent
sched-entity

- How to reflect child activity in parent?

- Use sched-entity
  - Sched_entity can be task
  - Sched_entity can be another runqueue

- runqueue schedules sched_entities
  - Each cfq_rq selects a sched_entity
  - Until reaching a task

- Task groups have 1 cfs_rq per CPU
  - Diagram on right is for 1 CPU
SHARE

- How to ensure fairness between groups of tasks
  - nice priority apply to the task group as whole
  - Prevent fork bomb process to starve single task process

- Share the weight between runqueues
  - A task group should not get more runtime than a single task with same priority

<table>
<thead>
<tr>
<th>1 task in group</th>
<th>3 tasks in group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half of the CPU</td>
<td>A third of the CPU</td>
</tr>
</tbody>
</table>
Update load and share

- Update load and weight of sched_entity
  - Split weight between sched_entities of task group

- Maintain a overall group load: update_tg_load_avg()
  - Periodic update for scalability

- Update sched_entities’ weight: update_cfs_group()  
  - Reweight the sched_entity

1 task on 1 CPU
2 tasks on 2 CPUs
3 tasks on 3 CPUs
Update load and share w/ migration

- Propagate task migration on branch
  - update metrics
  - update share

- Propagate utilization
  - Straight forward 1:1

- Propagate load
  - Just an estimate
  - Clamp results w/ simple rules
Flattening runqueue

- On going dev under eview

- All tasks are enqueued on root cfs rq
  - Inserted in root vruntime rb tree

- But still have a hierarchy
  - To maintain cgroup properties
  - To update per CPU share
CPUSET

- Apply a cpumask to a group of tasks
- Partition system in subsystem
  - Isolate group of CPUs
- Disable load balance
- Set load_balance level
CFS bandwidth

- Set a maximum runtime bandwidth to group of tasks

- Set bandwidth
  - cfs_period_us: period of monitoring and replenishment
  - cfs_quota_us: runtime for the group (can be > cfs_period_us)

- 1 CPU can use all bandwidth
uclamp

- Set some min/max utilization clamping
  - On system
  - On task (CFS or RT)
  - On a group of tasks

- Values are max aggregated

- Limited number of buckets
  - Limit the overhead
Task placement
Task placement

- Balance load between CPUs
  - Ensure fairness between tasks

- Use sched_domain / sched_group
  - Define policy: balance at wakeup?
  - Define interval: increase interval for larger cpumask span
  - Select which CPU will balance: 1 CPU per group

- When
  - @ wake up
  - Periodically
wakeup path

- Select a CPU when task wake up: `select_task_rq_fair()`

- Fast path
  - Looks at an idle CPU in LLC: `select_idle_sibling()`
  - A fully idle core
  - A sibling idle CPU
  - Any idle CPU

- Slow path
  - Looks for the idlest CPU @ largest sched_domain level: `find_idlest_cpu()`
  - Group with spare capacity
  - Significantly lower runnable load (only runnable task)
  - Significantly lower load (included blocked task)
Trigger load balance

- **Busy load balance**
  - Periodically check balance
  - Increase interval when busy

- **Idle load balance**
  - When there is idle CPU
  - When there is at least 2 tasks on the rq
  - When there is more than 1 busy CPU in the LLC
  - When task should run on a “bigger” CPU
  - CPU for all idle CPUs

- **Newly Idle load balance**
  - Done by idle_balance()
  - Only when there is an overload
  - Abort when balance is longer than average idle
  - Don’t delay task wake up
load balance

- **Update sched_group statistics**
  - Average runnable Load
  - Sum of capacity
  - nr_running tasks
  - ...

- **Classify groups:**
  - group_other = 0,
  - group_misfit_task,
  - group_imbalanced,
  - group_overloaded,

- **Update sched_domain statistics**
  - Will be used to calculate imbalance
Find busiest CPU

- Find busiest group in the domain
  - Imbalanced group because of pinned task
  - Group with asymmetric or misfit task case
  - Don’t creating new imbalance

- Find busiest queue in the group
  - Select queue with highest average load

- Except
  - Misfit task
  - Asymmetric packing
  - Single running task
Compute imbalance

- Compute imbalance
  - Estimate the load to move
  - Lot of exception including the fix_small_imbalance()

- Select waiting tasks
  - Not too large task
  - Try to play with pinned tasks problem

- Active migration
  - When failed to migrate waiting task
  - Asymmetric / misfit task case
Load balance rework

- Under review on LKML

- Changes:
  - Extend groups classification
    - group_has_spare
    - group_fully_busy
    - group_misfit_task
    - group_asym_capacity
    - group_imbalanced
    - group_overloaded
  - Define different type of migration
    - Migrate_load
    - migrate_util
    - migrate_task
    - migrate_misfit
  - Use load_avg instead of runnable_load_avg
Case of EAS/misfit task

- 2 modes:
  - Normal
  - Overutilized

- Normal mode:
  - CPUs have spare capacity
  - No always running task
  - Place task @ wakeup
  - find_energy_efficient_cpu()

- Overutilized / Misfit mode
  - Enable default load balance mechanism
Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org