Capacity Aware Scheduling

1. CPU Capacity

1.1 Introduction

Conventional, homogeneous SMP platforms are composed of purely identical CPUs. Heterogeneous platforms on the other hand are composed of CPUs with different performance characteristics - on such platforms, not all CPUs can be considered equal.

CPU capacity is a measure of the performance a CPU can reach, normalized against the most performant CPU in the system. Heterogeneous systems are also called asymmetric CPU capacity systems, as they contain CPUs of different capacities.

Disparity in maximum attainable performance (IOW in maximum CPU capacity) stems from two factors:

- not all CPUs may have the same microarchitecture (Âµarch).
- with Dynamic Voltage and Frequency Scaling (DVFS), not all CPUs may be physically able to attain the higher Operating Performance Points (OPP).

Arm big.LITTLE systems are an example of both. The big CPUs are more performance-oriented than the LITTLE ones (more pipeline stages, bigger caches, smarter predictors, etc), and can usually reach higher OPPs than the LITTLE ones can.

CPU performance is usually expressed in Millions of Instructions Per Second (MIPS), which can also be expressed as a given amount of instructions attainable per Hz, leading to:

```
capacity(cpu) = work_per_hz(cpu) * max_freq(cpu)
```

1.2 Scheduler terms

Two different capacity values are used within the scheduler. A CPU's `capacity_orig` is its maximum attainable capacity, i.e. its maximum attainable performance level. A CPU's `capacity` is its `capacity_orig` to which some loss of available performance (e.g. time spent handling IRQs) is subtracted.

Note that a CPU's `capacity` is solely intended to be used by the CFS class, while `capacity_orig` is class-agnostic. The rest of this document will use the term `capacity` interchangeably with `capacity_orig` for the sake of brevity.

1.3 Platform examples

1.3.1 Identical OPPs

Consider an hypothetical dual-core asymmetric CPU capacity system where

- work_per_hz(CPU0) = W
- work_per_hz(CPU1) = W/2
- all CPUs are running at the same fixed frequency

By the above definition of capacity:

- capacity(CPU0) = C
- capacity(CPU1) = C/2

To draw the parallel with Arm big.LITTLE, CPU0 would be a big while CPU1 would be a LITTLE.

With a workload that periodically does a fixed amount of work, you will get an execution trace like so:

```
CPU0 work ^
          |     ____                ____                ____
          |    |    |              |    |              |    |
          +----+----+----+----+----+----+----+----+----+----+-> time

CPU1 work ^
          |     _________           _________           ____
          |    |         |         |         |         |
          +----+----+----+----+----+----+----+----+----+----+-> time
```

CPU0 has the highest capacity in the system (C), and completes a fixed amount of work W in T units of time. On the other hand, CPU1 has half the capacity of CPU0, and thus only completes W/2 in T.

1.3.2 Different max OPPs

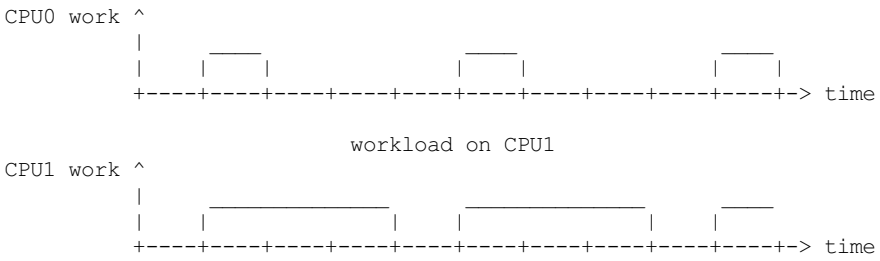
Usually, CPUs of different capacity values also have different maximum OPPs. Consider the same CPUs as above (i.e. same work_per_hz()) with:

- max_freq(CPU0) = F
- max_freq(CPU1) = 2/3 * F

This yields:

- capacity(CPU0) = C
- capacity(CPU1) = C/3

Executing the same workload as described in 1.3.1, which each CPU running at its maximum frequency results in:



1.4 Representation caveat

It should be noted that having a *single* value to represent differences in CPU performance is somewhat of a contentious point. The relative performance difference between two different Âµarchs could be X% on integer operations, Y% on floating point operations, Z% on branches, and so on. Still, results using this simple approach have been satisfactory for now.

2. Task utilization

2.1 Introduction

Capacity aware scheduling requires an expression of a task's requirements with regards to CPU capacity. Each scheduler class can express this differently, and while task utilization is specific to CFS, it is convenient to describe it here in order to introduce more generic concepts.

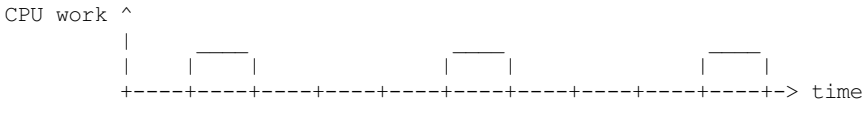
Task utilization is a percentage meant to represent the throughput requirements of a task. A simple approximation of it is the task's duty cycle, i.e.:

```
task_util(p) = duty_cycle(p)
```

On an SMP system with fixed frequencies, 100% utilization suggests the task is a busy loop. Conversely, 10% utilization hints it is a small periodic task that spends more time sleeping than executing. Variable CPU frequencies and asymmetric CPU capacities complexify this somewhat; the following sections will expand on these.

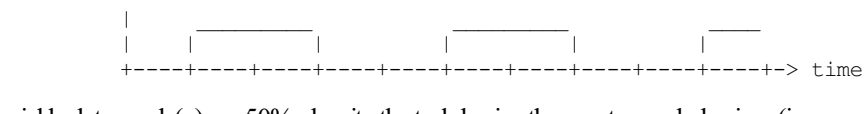
2.2 Frequency invariance

One issue that needs to be taken into account is that a workload's duty cycle is directly impacted by the current OPP the CPU is running at. Consider running a periodic workload at a given frequency F:



This yields duty_cycle(p) == 25%.

Now, consider running the *same* workload at frequency F/2:



This yields duty_cycle(p) == 50%, despite the task having the exact same behaviour (i.e. executing the same amount of work) in both executions.

The task utilization signal can be made frequency invariant using the following formula:

```
task_util_freq_inv(p) = duty_cycle(p) * (curr_frequency(cpu) / max_frequency(cpu))
```

Applying this formula to the two examples above yields a frequency invariant task utilization of 25%.

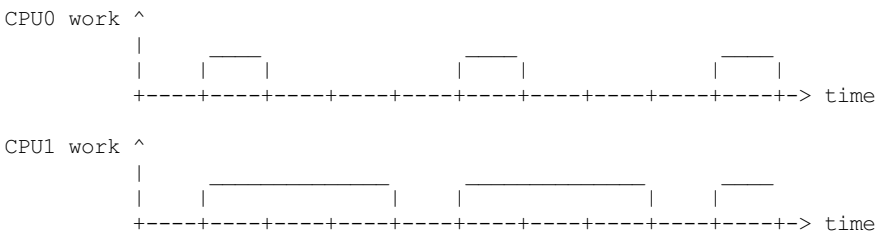
2.3 CPU invariance

CPU capacity has a similar effect on task utilization in that running an identical workload on CPUs of different capacity values will yield different duty cycles.

Consider the system described in 1.3.2., i.e.:

```
- capacity(CPU0) = C
- capacity(CPU1) = C/3
```

Executing a given periodic workload on each CPU at their maximum frequency would result in:



IOW,

- duty_cycle(p) == 25% if p runs on CPU0 at its maximum frequency
- duty_cycle(p) == 75% if p runs on CPU1 at its maximum frequency

The task utilization signal can be made CPU invariant using the following formula:

```
task_util_cpu_inv(p) = duty_cycle(p) * (capacity(cpu) / max_capacity)
```

with `max_capacity` being the highest CPU capacity value in the system. Applying this formula to the above example above yields a CPU invariant task utilization of 25%.

2.4 Invariant task utilization

Both frequency and CPU invariance need to be applied to task utilization in order to obtain a truly invariant signal. The pseudo-formula for a task utilization that is both CPU and frequency invariant is thus, for a given task p:

```
                                  curr_frequency(cpu)   capacity(cpu)
task_util_inv(p) = duty_cycle(p) * ------------------- * -------------
                                  max_frequency(cpu)    max_capacity
```

In other words, invariant task utilization describes the behaviour of a task as if it were running on the highest-capacity CPU in the system, running at its maximum frequency.

Any mention of task utilization in the following sections will imply its invariant form.

2.5 Utilization estimation

Without a crystal ball, task behaviour (and thus task utilization) cannot accurately be predicted the moment a task first becomes runnable. The CFS class maintains a handful of CPU and task signals based on the Per-Entity Load Tracking (PELT) mechanism, one of those yielding an *average* utilization (as opposed to instantaneous).

This means that while the capacity aware scheduling criteria will be written considering a "true" task utilization (using a crystal ball), the implementation will only ever be able to use an estimator thereof.

3. Capacity aware scheduling requirements

3.1 CPU capacity

Linux cannot currently figure out CPU capacity on its own, this information thus needs to be handed to it. Architectures must define arch_scale_cpu_capacity() for that purpose.

The arm and arm64 architectures directly map this to the arch_topology driver CPU scaling data, which is derived from the capacity-dmips-mhz CPU binding; see Documentation/devicetree/bindings/arm/cpu-capacity.txt.

3.2 Frequency invariance

As stated in 2.2, capacity-aware scheduling requires a frequency-invariant task utilization. Architectures must define arch_scale_freq_capacity(cpu) for that purpose.

Implementing this function requires figuring out at which frequency each CPU have been running at. One way to implement this is to leverage hardware counters whose increment rate scale with a CPU's current frequency (APERF/MPERF on x86, AMU on arm64). Another is to directly hook into cpufreq frequency transitions, when the kernel is aware of the switched-to frequency (also employed by arm/arm64).

4. Scheduler topology

During the construction of the sched domains, the scheduler will figure out whether the system exhibits asymmetric CPU capacities. Should that be the case:

- The sched_asym_cpucapacity static key will be enabled.
- The SD_ASYM_CPUCAPACITY_FULL flag will be set at the lowest sched_domain level that spans all unique CPU capacity values.

- The SD_ASYM_CPUCAPACITY flag will be set for any sched_domain that spans CPUs with any range of asymmetry.

The sched_asym_cpucapacity static key is intended to guard sections of code that cater to asymmetric CPU capacity systems. Do note however that said key is *system-wide*. Imagine the following setup using cpusets:

```
capacity    C/2          C

          _________   _________
         /         \ /         \
CPUs     0  1  2  3  4  5  6  7
         \__/  \_______________/
cpusets   cs0         cs1
```

Which could be created via:

```
mkdir /sys/fs/cgroup/cpuset/cs0
echo 0-1 > /sys/fs/cgroup/cpuset/cs0/cpuset.cpus
echo 0 > /sys/fs/cgroup/cpuset/cs0/cpuset.mems

mkdir /sys/fs/cgroup/cpuset/cs1
echo 2-7 > /sys/fs/cgroup/cpuset/cs1/cpuset.cpus
echo 0 > /sys/fs/cgroup/cpuset/cs1/cpuset.mems

echo 0 > /sys/fs/cgroup/cpuset/cpuset.sched_load_balance
```

Since there *is* CPU capacity asymmetry in the system, the sched_asym_cpucapacity static key will be enabled. However, the sched_domain hierarchy of CPUs 0-1 spans a single capacity value: SD_ASYM_CPUCAPACITY isn't set in that hierarchy, it describes an SMP island and should be treated as such.

Therefore, the 'canonical' pattern for protecting codepaths that cater to asymmetric CPU capacities is to:

- Check the sched_asym_cpucapacity static key
- If it is enabled, then also check for the presence of SD_ASYM_CPUCAPACITY in the sched_domain hierarchy (if relevant, i.e. the codepath targets a specific CPU or group thereof)

5. Capacity aware scheduling implementation

5.1 CFS

5.1.1 Capacity fitness

The main capacity scheduling criterion of CFS is:

```
task_util(p) < capacity(task_cpu(p))
```

This is commonly called the capacity fitness criterion, i.e. CFS must ensure a task "fits" on its CPU. If it is violated, the task will need to achieve more work than what its CPU can provide: it will be CPU-bound.

Furthermore, uclamp lets userspace specify a minimum and a maximum utilization value for a task, either via sched_setattr() or via the cgroup interface (see Documentation/admin-guide/cgroup-v2.rst). As its name imply, this can be used to clamp task_util() in the previous criterion.

5.1.2 Wakeup CPU selection

CFS task wakeup CPU selection follows the capacity fitness criterion described above. On top of that, uclamp is used to clamp the task utilization values, which lets userspace have more leverage over the CPU selection of CFS tasks. IOW, CFS wakeup CPU selection searches for a CPU that satisfies:

```
clamp(task_util(p), task_uclamp_min(p), task_uclamp_max(p)) < capacity(cpu)
```

By using uclamp, userspace can e.g. allow a busy loop (100% utilization) to run on any CPU by giving it a low uclamp.max value. Conversely, it can force a small periodic task (e.g. 10% utilization) to run on the highest-performance CPUs by giving it a high uclamp.min value.

> **Note**
>
> Wakeup CPU selection in CFS can be eclipsed by Energy Aware Scheduling (EAS), which is described in Documentation/scheduler/sched-energy.rst.

5.1.3 Load balancing

A pathological case in the wakeup CPU selection occurs when a task rarely sleeps, if at all - it thus rarely wakes up, if at all. Consider:

```
w == wakeup event
```

```
capacity(CPU0) = C
capacity(CPU1) = C / 3

                          workload on CPU0
CPU work ^
         |     _________          _________          ____
         |    |         |        |         |        |
         +----+----+----+----+---+----+----+----+---+----+-> time
              w                  w                  w

                          workload on CPU1
CPU work ^
         |     _____________________________________________
         |    |
         +----+----+----+----+----+----+----+----+----+----+->
              w
```

This workload should run on CPU0, but if the task either:

- was improperly scheduled from the start (inaccurate initial utilization estimation)
- was properly scheduled from the start, but suddenly needs more processing power

then it might become CPU-bound, IOW `task_util(p) > capacity(task_cpu(p))`; the CPU capacity scheduling criterion is violated, and there may not be any more wakeup event to fix this up via wakeup CPU selection.

Tasks that are in this situation are dubbed "misfit" tasks, and the mechanism put in place to handle this shares the same name. Misfit task migration leverages the CFS load balancer, more specifically the active load balance part (which caters to migrating currently running tasks). When load balance happens, a misfit active load balance will be triggered if a misfit task can be migrated to a CPU with more capacity than its current one.

5.2 RT

5.2.1 Wakeup CPU selection

RT task wakeup CPU selection searches for a CPU that satisfies:

```
task_uclamp_min(p) <= capacity(task_cpu(cpu))
```

while still following the usual priority constraints. If none of the candidate CPUs can satisfy this capacity criterion, then strict priority based scheduling is followed and CPU capacities are ignored.

5.3 DL

5.3.1 Wakeup CPU selection

DL task wakeup CPU selection searches for a CPU that satisfies:

```
task_bandwidth(p) < capacity(task_cpu(p))
```

while still respecting the usual bandwidth and deadline constraints. If none of the candidate CPUs can satisfy this capacity criterion, then the task will remain on its current CPU.