Energy Aware Scheduling

1. Introduction

Energy Aware Scheduling (or EAS) gives the scheduler the ability to predict the impact of its decisions on the energy consumed by CPUs. EAS relies on an Energy Model (EM) of the CPUs to select an energy efficient CPU for each task, with a minimal impact on throughput. This document aims at providing an introduction on how EAS works, what are the main design decisions behind it, and details what is needed to get it to run.

Before going any further, please note that at the time of writing:

```
/!\ EAS does not support platforms with symmetric CPU topologies /!\
```

EAS operates only on heterogeneous CPU topologies (such as Arm big.LITTLE) because this is where the potential for saving energy through scheduling is the highest.

The actual EM used by EAS is _not_ maintained by the scheduler, but by a dedicated framework. For details about this framework and what it provides, please refer to its documentation (see Documentation/power/energy-model.rst).

2. Background and Terminology

To make it clear from the start:

- energy = [joule] (resource like a battery on powered devices)
- power = energy/time = [joule/second] = [watt]

The goal of EAS is to minimize energy, while still getting the job done. That is, we want to maximize:

```
performance [inst/s]
--------------------
    power [W]
```

which is equivalent to minimizing:

```
energy [J]
-----------
instruction
```

while still getting 'good' performance. It is essentially an alternative optimization objective to the current performance-only objective for the scheduler. This alternative considers two objectives: energy-efficiency and performance.

The idea behind introducing an EM is to allow the scheduler to evaluate the implications of its decisions rather than blindly applying energy-saving techniques that may have positive effects only on some platforms. At the same time, the EM must be as simple as possible to minimize the scheduler latency impact.

In short, EAS changes the way CFS tasks are assigned to CPUs. When it is time for the scheduler to decide where a task should run (during wake-up), the EM is used to break the tie between several good CPU candidates and pick the one that is predicted to yield the best energy consumption without harming the system's throughput. The predictions made by EAS rely on specific elements of knowledge about the platform's topology, which include the 'capacity' of CPUs, and their respective energy costs.

3. Topology information

EAS (as well as the rest of the scheduler) uses the notion of 'capacity' to differentiate CPUs with different computing throughput. The 'capacity' of a CPU represents the amount of work it can absorb when running at its highest frequency compared to the most capable CPU of the system. Capacity values are normalized in a 1024 range, and are comparable with the utilization signals of tasks and CPUs computed by the Per-Entity Load Tracking (PELT) mechanism. Thanks to capacity and utilization values, EAS is able to estimate how big/busy a task/CPU is, and to take this into consideration when evaluating performance vs energy trade-offs. The capacity of CPUs is provided via arch-specific code through the arch_scale_cpu_capacity() callback.

The rest of platform knowledge used by EAS is directly read from the Energy Model (EM) framework. The EM of a platform is composed of a power cost table per 'performance domain' in the system (see Documentation/power/energy-model.rst for futher details about performance domains).

The scheduler manages references to the EM objects in the topology code when the scheduling domains are built, or re-built. For each root domain (rd), the scheduler maintains a singly linked list of all performance domains intersecting the current rd->span. Each node in the list contains a pointer to a struct em_perf_domain as provided by the EM framework.

The lists are attached to the root domains in order to cope with exclusive cpuset configurations. Since the boundaries of exclusive cpusets do not necessarily match those of performance domains, the lists of different root domains can contain duplicate elements.

Example 1.

Let us consider a platform with 12 CPUs, split in 3 performance domains (pd0, pd4 and pd8), organized as follows:

```
CPUs:   0 1 2 3 4 5 6 7 8 9 10 11
PDs:   |--pd0--|--pd4--|---pd8---|
RDs:   |----rd1----|-----rd2-----|
```

Now, consider that userspace decided to split the system with two exclusive cpusets, hence creating two independent root domains, each containing 6 CPUs. The two root domains are denoted rd1 and rd2 in the above figure. Since pd4 intersects with both rd1 and rd2, it will be present in the linked list '->pd' attached to each of them:

- rd1->pd: pd0 -> pd4
- rd2->pd: pd4 -> pd8

Please note that the scheduler will create two duplicate list nodes for pd4 (one for each list). However, both just hold a pointer to the same shared data structure of the EM framework.

Since the access to these lists can happen concurrently with hotplug and other things, they are protected by RCU, like the rest of topology structures manipulated by the scheduler.

EAS also maintains a static key (sched_energy_present) which is enabled when at least one root domain meets all conditions for EAS to start. Those conditions are summarized in Section 6.

4. Energy-Aware task placement

EAS overrides the CFS task wake-up balancing code. It uses the EM of the platform and the PELT signals to choose an energy-efficient target CPU during wake-up balance. When EAS is enabled, select_task_rq_fair() calls find_energy_efficient_cpu() to do the placement decision. This function looks for the CPU with the highest spare capacity (CPU capacity - CPU utilization) in each performance domain since it is the one which will allow us to keep the frequency the lowest. Then, the function checks if placing the task there could save energy compared to leaving it on prev_cpu, i.e. the CPU where the task ran in its previous activation.

find_energy_efficient_cpu() uses compute_energy() to estimate what will be the energy consumed by the system if the waking task was migrated. compute_energy() looks at the current utilization landscape of the CPUs and adjusts it to 'simulate' the task migration. The EM framework provides the em_pd_energy() API which computes the expected energy consumption of each performance domain for the given utilization landscape.

An example of energy-optimized task placement decision is detailed below.

Example 2.

Let us consider a (fake) platform with 2 independent performance domains composed of two CPUs each. CPU0 and CPU1 are little CPUs; CPU2 and CPU3 are big.

The scheduler must decide where to place a task P whose util_avg = 200 and prev_cpu = 0.

The current utilization landscape of the CPUs is depicted on the graph below. CPUs 0-3 have a util_avg of 400, 100, 600 and 500 respectively Each performance domain has three Operating Performance Points (OPPs). The CPU capacity and power cost associated with each OPP is listed in the Energy Model table. The util_avg of P is shown on the figures below as 'PP':

```
CPU util.
  1024                 - - - - - - -              Energy Model
                                           +-----------+-------------+
                                           |  Little   |     Big     |
   768                 =============       +-----+-----+------+------+
                                           | Cap | Pwr | Cap  | Pwr  |
                                           +-----+-----+------+------+
   512  ===========    - ##- - - - -       | 170 | 50  | 512  | 400  |
                         ##     ##         | 341 | 150 | 768  | 800  |
   341  -PP - - - -      ##     ##         | 512 | 300 | 1024 | 1700 |
         PP              ##     ##         +-----+-----+------+------+
   170  -## - - - -      ##     ##
         ##     ##       ##     ##
       ------------    -------------
        CPU0   CPU1     CPU2   CPU3

  Current OPP: =====       Other OPP: - - -     util_avg (100 each): ##
```

find_energy_efficient_cpu() will first look for the CPUs with the maximum spare capacity in the two performance domains. In this example, CPU1 and CPU3. Then it will estimate the energy of the system if P was placed on either of them, and check if that would save some energy compared to leaving P on CPU0. EAS assumes that OPPs follow utilization (which is coherent with the behaviour of the schedutil CPUFreq governor, see Section 6. for more details on this topic).

Case 1. P is migrated to CPU1:

```
1024                 - - - - - - -

                                          Energy calculation:
 768                 =============         * CPU0: 200 / 341 * 150 = 88
                                           * CPU1: 300 / 341 * 150 = 131
```

```
                                       * CPU2: 600 / 768 * 800 = 625
 512  - - - - - -    - ##- - - - -     * CPU3: 500 / 768 * 800 = 520
                       ##      ##         => total_energy = 1364
 341  ===========      ##      ##
              PP       ##      ##
 170  -## - - PP-      ##      ##
       ##     ##       ##      ##
     ------------    -------------
      CPU0   CPU1     CPU2   CPU3
```

Case 2. P is migrated to CPU3:

```
1024                 - - - - - - -

                                      Energy calculation:
 768                 =============     * CPU0: 200 / 341 * 150 = 88
                                       * CPU1: 100 / 341 * 150 = 43
                              PP       * CPU2: 600 / 768 * 800 = 625
 512  - - - - - -    - ##- - -PP -     * CPU3: 700 / 768 * 800 = 729
                       ##      ##         => total_energy = 1485
 341  ===========      ##      ##
                       ##      ##
 170  -## - - - -      ##      ##
       ##     ##       ##      ##
     ------------    -------------
      CPU0   CPU1     CPU2   CPU3
```

Case 3. P stays on prev_cpu / CPU 0:

```
1024                 - - - - - - -

                                      Energy calculation:
 768                 =============     * CPU0: 400 / 512 * 300 = 234
                                       * CPU1: 100 / 512 * 300 = 58
                                       * CPU2: 600 / 768 * 800 = 625
 512  ===========    - ##- - - - -     * CPU3: 500 / 768 * 800 = 520
                       ##      ##         => total_energy = 1437
 341  -PP - - - -      ##      ##
       PP              ##      ##
 170  -## - - - -      ##      ##
       ##     ##       ##      ##
     ------------    -------------
      CPU0   CPU1     CPU2   CPU3
```

From these calculations, the Case 1 has the lowest total energy. So CPU 1 is be the best candidate from an energy-efficiency standpoint.

Big CPUs are generally more power hungry than the little ones and are thus used mainly when a task doesn't fit the littles. However, little CPUs aren't always necessarily more energy-efficient than big CPUs. For some systems, the high OPPs of the little CPUs can be less energy-efficient than the lowest OPPs of the bigs, for example. So, if the little CPUs happen to have enough utilization at a specific point in time, a small task waking up at that moment could be better of executing on the big side in order to save energy, even though it would fit on the little side.

And even in the case where all OPPs of the big CPUs are less energy-efficient than those of the little, using the big CPUs for a small task might still, under specific conditions, save energy. Indeed, placing a task on a little CPU can result in raising the OPP of the entire performance domain, and that will increase the cost of the tasks already running there. If the waking task is placed on a big CPU, its own execution cost might be higher than if it was running on a little, but it won't impact the other tasks of the little CPUs which will keep running at a lower OPP. So, when considering the total energy consumed by CPUs, the extra cost of running that one task on a big core can be smaller than the cost of raising the OPP on the little CPUs for all the other tasks.

The examples above would be nearly impossible to get right in a generic way, and for all platforms, without knowing the cost of running at different OPPs on all CPUs of the system. Thanks to its EM-based design, EAS should cope with them correctly without too many troubles. However, in order to ensure a minimal impact on throughput for high-utilization scenarios, EAS also implements another mechanism called 'over-utilization'.

5. Over-utilization

From a general standpoint, the use-cases where EAS can help the most are those involving a light/medium CPU utilization. Whenever long CPU-bound tasks are being run, they will require all of the available CPU capacity, and there isn't much that can be done by the scheduler to save energy without severly harming throughput. In order to avoid hurting performance with EAS, CPUs are flagged as 'over-utilized' as soon as they are used at more than 80% of their compute capacity. As long as no CPUs are over-utilized in a root domain, load balancing is disabled and EAS overridess the wake-up balancing code. EAS is likely to load the most energy efficient CPUs of the system more than the others if that can be done without harming throughput. So, the load-balancer is disabled to prevent it from breaking the energy-efficient task placement found by EAS. It is safe to do so when the system isn't overutilized since being below the 80% tipping point implies that:

a. there is some idle time on all CPUs, so the utilization signals used by EAS are likely to accurately represent the 'size' of the various tasks in the system;
b. all tasks should already be provided with enough CPU capacity, regardless of their nice values;
c. since there is spare capacity all tasks must be blocking/sleeping regularly and balancing at wake-up is sufficient.

As soon as one CPU goes above the 80% tipping point, at least one of the three assumptions above becomes incorrect. In this scenario, the 'overutilized' flag is raised for the entire root domain, EAS is disabled, and the load-balancer is re-enabled. By doing so, the scheduler falls back onto load-based algorithms for wake-up and load balance under CPU-bound conditions. This provides a better respect of the nice values of tasks.

Since the notion of overutilization largely relies on detecting whether or not there is some idle time in the system, the CPU capacity 'stolen' by higher (than CFS) scheduling classes (as well as IRQ) must be taken into account. As such, the detection of overutilization accounts for the capacity used not only by CFS tasks, but also by the other scheduling classes and IRQ.

6. Dependencies and requirements for EAS

Energy Aware Scheduling depends on the CPUs of the system having specific hardware properties and on other features of the kernel being enabled. This section lists these dependencies and provides hints as to how they can be met.

6.1 - Asymmetric CPU topology

As mentioned in the introduction, EAS is only supported on platforms with asymmetric CPU topologies for now. This requirement is checked at run-time by looking for the presence of the SD_ASYM_CPUCAPACITY_FULL flag when the scheduling domains are built.

See Documentation/scheduler/sched-capacity.rst for requirements to be met for this flag to be set in the sched_domain hierarchy.

Please note that EAS is not fundamentally incompatible with SMP, but no significant savings on SMP platforms have been observed yet. This restriction could be amended in the future if proven otherwise.

6.2 - Energy Model presence

EAS uses the EM of a platform to estimate the impact of scheduling decisions on energy. So, your platform must provide power cost tables to the EM framework in order to make EAS start. To do so, please refer to documentation of the independent EM framework in Documentation/power/energy-model.rst.

Please also note that the scheduling domains need to be re-built after the EM has been registered in order to start EAS.

EAS uses the EM to make a forecasting decision on energy usage and thus it is more focused on the difference when checking possible options for task placement. For EAS it doesn't matter whether the EM power values are expressed in milli-Watts or in an 'abstract scale'.

6.3 - Energy Model complexity

The task wake-up path is very latency-sensitive. When the EM of a platform is too complex (too many CPUs, too many performance domains, too many performance states, ...), the cost of using it in the wake-up path can become prohibitive. The energy-aware wake-up algorithm has a complexity of:

$$C = Nd * (Nc + Ns)$$

with: Nd the number of performance domains; Nc the number of CPUs; and Ns the total number of OPPs (ex: for two perf. domains with 4 OPPs each, Ns = 8).

A complexity check is performed at the root domain level, when scheduling domains are built. EAS will not start on a root domain if its C happens to be higher than the completely arbitrary EM_MAX_COMPLEXITY threshold (2048 at the time of writing).

If you really want to use EAS but the complexity of your platform's Energy Model is too high to be used with a single root domain, you're left with only two possible options:

1. split your system into separate, smaller, root domains using exclusive cpusets and enable EAS locally on each of them. This option has the benefit to work out of the box but the drawback of preventing load balance between root domains, which can result in an unbalanced system overall;
2. submit patches to reduce the complexity of the EAS wake-up algorithm, hence enabling it to cope with larger EMs in reasonable time.

6.4 - Schedutil governor

EAS tries to predict at which OPP will the CPUs be running in the close future in order to estimate their energy consumption. To do so, it is assumed that OPPs of CPUs follow their utilization.

Although it is very difficult to provide hard guarantees regarding the accuracy of this assumption in practice (because the hardware might not do what it is told to do, for example), schedutil as opposed to other CPUFreq governors at least _requests_ frequencies calculated using the utilization signals. Consequently, the only sane governor to use together with EAS is schedutil, because it is the only one providing some degree of consistency between frequency requests and energy predictions.

Using EAS with any other governor than schedutil is not supported.

6.5 Scale-invariant utilization signals

In order to make accurate prediction across CPUs and for all performance states, EAS needs frequency-invariant and CPU-invariant PELT signals. These can be obtained using the architecture-defined arch_scale{cpu,freq}_capacity() callbacks.

Using EAS on a platform that doesn't implement these two callbacks is not supported.

6.6 Multithreading (SMT)

EAS in its current form is SMT unaware and is not able to leverage multithreaded hardware to save energy. EAS considers threads as independent CPUs, which can actually be counter-productive for both performance and energy.

EAS on SMT is not supported.