Revision Information

The following revisions have been made to this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>26 September 2016</td>
<td>1.1</td>
<td>Non-Confidential</td>
<td>LSK 16.09 release</td>
</tr>
<tr>
<td>14 November 2016</td>
<td>1.1a</td>
<td>Non-Confidential</td>
<td>LSK 16.10 release</td>
</tr>
</tbody>
</table>

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM® in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

Confidentiality Status

This document is Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is final, that is for a developed product.

Web Address

http://www.arm.com
7.1  An Overview of the LISA Toolkit ........................................................................................................31
  Hardware Abstraction Layer ..................................................................................................................32
  Data Collection and Analysis ................................................................................................................32
  Interactive Test and Analysis ................................................................................................................33
  Automated Tests ..................................................................................................................................33
7.2  Integrating a board into LISA ............................................................................................................33
7.3  Building an Energy Model .................................................................................................................34
7.4  Verifying EAS functionality ................................................................................................................34
  EAS acceptance tests ..............................................................................................................................34
  Capacity capping testing .........................................................................................................................36
  Analyze test failures ...............................................................................................................................37
  Evaluate per task behavior ......................................................................................................................38
7.5  EAS test coverage ...............................................................................................................................40
8    Glossary .................................................................................................................................................41
9    Appendix ...............................................................................................................................................43
9.1  Example DTS file ................................................................................................................................43
1 Overview

1.1 Motivation for EAS

Energy Aware Scheduling (EAS) was conceived as a set of techniques to:

Reduce fragmentation and redundancy in the power-performance related subsystems of the Linux kernel

Subsystems such as cpuidle (the primary idle management system for application processors), cpufreq (the Dynamic Voltage and Frequency Scaling management subsystem for application processors) and the kernel task scheduler all maintained similar system heuristics but used them in independent ways. This promoted a dynamic in which desired power-performance objectives were haphazardly serviced with these subsystems often getting in each other's way. For example, cpufreq would change the operating frequency of a processor core unbeknown to the scheduler which would then incorrectly account for that processing element's load potentially resulting in sub-optimal task placement choices.

Allow for more centralized and holistic power-performance management via the kernel's task scheduler

Related to the previous point, rather than allow multiple subsystems to each attempt driving power-performance independently, often leading to a chaotic outcome, the essential requirement was to drive relevant subsystems from a central, informed point in the kernel. The task scheduler was uniquely provisioned to know the compute requirements of the tasks it scheduled on various processing elements. As such, it made sense to get the scheduler to place requests on key subsystems like cpufreq via well-defined interfaces and for such subsystems to be dumb clients which left policy considerations to the scheduler. This was unlike the traditional approach where subsystems like cpufreq maintained both policy as well as mechanics in an insular manner. As an example, with EAS the intent was to reduce cpufreq to just platform specific mechanics and leave policy to the scheduler.

Truly enable the consideration of energy impact in power-performance management

The traditional approach to power-performance management in UNIX derived systems has been to introspectively analyse system utilisation and attempt to gauge the consequent energy impact. This has proven to be very fragile and often incorrect. For example, it is extremely difficult to establish which action is more energy efficient for a given system and to what extent: spreading tasks to more processing elements at lower operating points or packing those tasks to fewer processing elements at a higher operating point. The cleanest way to solve such problems is to stop guessing and rely on a priori inputs to the system. For the example in question, if it was known what the energy sacrifice would be to run processing elements at the supported operating points, the decision to choose one task placement dynamic over another would be far more straightforward. EAS introduced a formal specification of a system's power-performance capability via a system specific energy model. This also had the welcome side effect of providing a better abstraction for the system topology and promoted a uniform design theoretically capable of servicing different system topologies.
Localize the tuning surface

EAS' primary target is mobile consumer systems such as smart phones. The software stacks on such systems place multiple orthogonal requests on the kernel and servicing these requests efficiently is a hard problem. Fundamentally, the system is required to service the following types of operations:

- Throughput oriented operation (e.g. multi-threaded, high intensity workloads like performance benchmarks)
- Energy efficient operation (e.g. low to medium intensity workloads like audio playback or episodes of browser page rendering)
- Latency sensitive operation (e.g. UI workloads like flicking through Android's recently used applications)

The traditional approach to servicing these types of operations optimally is to exploit miscellaneous tunables exposed by the multiple power-performance subsystems. However, the number of tunables provided by each power performance subsystem made it extremely difficult to cleanly express and achieve a desired system wide power-performance objective. For example, the task scheduler, cpufreq and cpuidle all provided their own set of tunables and it was difficult to establish how one subsystem's tunables influenced the other subsystems or which combination of tunable values belonging to a set of subsystems would be needed to target one of the operations listed above.

Finally, no amount of system specific awareness can help the kernel introspectively identify and service some of the above operations – especially latency sensitive operations. These simply do require scenario specific inputs to the kernel. For example: explicit triggers to the kernel when a mobile phone screen is touched. The problem with all of the above was that no single, well specified interface was specified which could permit scenario aware software layers (like Android middleware) to request specific compute for specific tasks. EAS's SchedTune component provides such an interface.

Make energy saving a primary consideration rather than an afterthought

Establishing the exact nature of the power-performance capabilities of a system has always been approached indirectly and to a large extent superficially by system software developers tasked with supporting a platform. For example, cpufreq and cpuidle driver writers would often enough copy tunable values for their platform specific driver from existing drivers and leave things at that. This meant that energy consequences were only really considered when bad things happened at awkward stages in product development life cycles. The need of the hour was to flip things around such that energy awareness was considered right from the beginning. To make that possible, EAS specifies a methodology for developing an energy model for a system as the first step and then mandates the use of that model for all decision making. This promotes energy aware design and more informed system software development with developers interacting more closely with system integrators and power architects than ever before.

These motivations that led to the development of EAS emerged from multiple endeavours to understand and improve the Linux power – performance landscape for ARM's big.LITTLE designs. Over the years, these endeavours resulted in a lot of interactions with the open source community in multiple forums as well as a wide spectrum of ARM
partners. EAS as it stands today is the result of knowledge and experience gathered from these interactions.

This document aims to describe the architecture of EAS as present in the EAS product codeline. The intended audience for this document are developers interested in porting, tuning and evaluation of the product codeline.

1.2 Application of EAS

The mainline Completely Fair Scheduler (CFS) scheduler class implements a throughput oriented task placement policy. EAS adds an energy based policy to this scheduler class which permits energy saving without harming throughput by managing the spare capacity of CPUs intelligently. EAS and the original CFS policy operate in different ranges of system utilization.

EAS development targeted dual-cluster big.LITTLE systems with non-overlapping power/performance curves for the two cpu core types and per-cluster and/or per-cpu power-down cpu-idle states. Such a topology allows EAS to clearly show its advantages over the vanilla mainline CFS scheduler for a wide range of workloads.

The current EAS implementation does not support Symmetric Multi-Threading (SMT) nor Non-Uniform Memory Access (NUMA) architectures.

1.3 Scope of the document

This software release is suitable for product evaluation of the Energy-Aware Scheduler. It represents a hardened version of the ongoing open-source EAS development. The open-source EAS project for mainline Linux takes place on LKML mailing list. Its progress is depicted on the following ARM developer website:

https://developer.arm.com/open-source/energy-aware-scheduling

This document contains information about where the source code is hosted in chapter “Source Code”, explains the building blocks and the functionality of EAS in chapter “Functionality” and how to evaluate a new EAS integration on a new device in chapter “Integration”.
1.4 EAS Overview

EAS extends a few different subsystems present in the kernel. A major part of EAS is located in the file: `kernel/sched/fair.c` and is the algorithm responsible for task placement decisions. This module constructs necessary structures containing energy metrics which are used for calculating energy efficiency. The code extends commonly used CFS scheduling policy and does not touch other policies. Some of the existing features in CFS have been made energy aware by factoring in the possible energy cost of scheduling tasks and managing the CPUs the tasks run on. EAS therefore impacts load balancing and task packing decisions. A key realisation was that it only makes sense to bias task placement in favour of energy efficient operation when there is spare CPU capacity available. In the absence of spare capacity, the system is usually in a state where throughput is the primary need. Intentionally biasing towards energy efficient operation in such cases could compromise throughput.

A set of heuristics have been introduced which enable under-utilized systems to run tasks in an energy efficient manner. When the system is over-utilized, EAS is effectively disabled, with the rules for task placement falling back to conventional CFS rules. EAS considers the system over-utilized when even one CPU’s utilization is above a certain limit. This dynamic helps to spread work across CPUs when there is a real need for throughput while also giving the scheduler a chance to optimize task placement for energy efficiency when there is spare capacity available.
Enabling the scheduler to judge the power and performance requirements of tasks needs the introduction of new data structures centred around the concept of an energy model. Therefore, a common mechanism has been introduced for providing an energy model to the scheduler. This solution aspires to be universally beneficial for all platforms and use-cases.

The energy model data consists of:

- power consumption for each supported P-state (this is the DVFS Operating Performance Point (OPP) which is a tuple of frequency and the associated voltage)
- power consumption for each C-state (the idle power management state)
- wake-up energy cost for each C-state

The model only contains data for CPUs and clusters. The cluster maintenance energy, which can vary depending on specific architecture, is added to the energy related to each CPU to have accurate approximations.

Energy model data is provided to the kernel using the Flattened Device Tree (FDT) mechanism. Extensions to the FDT specification have been made that enable the expression of energy model data (see section Device Tree for further detail). A system specific FDT description is given to the kernel via a so called FDT blob as per the usual practice prevalent in Linux. The energy model data within the device tree is given to the scheduler via function present in kernel/sched/energy.c. Architecture specific code relevant to EAS includes a shim layer that builds a system topology for the scheduler. There are also some extensions for the main EAS algorithm such as FIE (Frequency Invariant Engine) and CIE (CPU Invariant Engine). These extensions help normalize the load calculations made by the scheduler and make these calculations invariant of the CPU frequency and micro-architecture. This scale-invariance improves the estimation of CPU utilization by factoring in the microarchitecture differences between CPUs as well as the current CPU frequency. Suitable scaling correction factors are provided for more accurate load-tracking.

Per-task load tracking in Linux (and by extension also EAS) is implemented using the Per-Entity Load Tracking (PELT) technique by the CFS scheduler class. The EAS product codeline introduces another load tracking mechanism known as Window Assist Load Tracking (WALT). WALT is used selectively in the product codeline at present. This is because when compared with PELT, WALT promotes faster reaction times when the behaviour of tasks changes. Faster reaction time is a key requirement for Android. WALT uses periodic calculations that are synchronized across all of the run queues, attempting to track the behaviour of all scheduling classes (while PELT is focused only on a single class - CFS). A big advantage of this approach is that the decisions can be made based on the information about the full state of the running system. The drawback is additional locking complexity and some additional delays in other pathways.

Energy aware task placement decisions require the scheduler to estimate the energy impact in case of scheduling a specific task on a specific cpu. Sometimes it could be more energy efficient to wake up another cpu rather than alter the P-state of the current CPU where the given task runs. This calculation adds a small latency overhead to the scheduler. This feature is used at target run queue selection time for the task. At this point a decision needs to be taken to choose one of two possible pathways: energy efficient pathway or going to a sibling CPU.
As mentioned previously, EAS is primarily targeted at promoting energy efficient operation when there is spare capacity present in the system. To assist with that style of operation, a new ‘utilization’ metric was introduced. The utilization metric, in addition to the load metric, simplifies energy-aware scheduling decisions. A CPU’s utilization corresponds to the CPU’s capacity. Therefore, CPU utilization can be compared with the currently available CPU capacity for CFS tasks.

EAS has a notion of over-utilization. The diagram below shows sites where the scheduler flags that the system is over-utilized.

Figure 2 Task wakeup path modifications. (The yellow dots represent EAS functionality)
Figure 3 Places where the scheduler potentially marks the system as over-utilized

When one CPU is over-utilized, the whole system is considered as over-utilized. In this case the scheduler opts for a task spreading dynamic via conventional load balance pathways (as opposed to the new energy aware pathway). Note that the latter is implemented by the `energy_aware_wake_cpu()` function which promotes a task packing dynamic aimed at keeping CPUs idle).

The modification of the load balance subsystem is shown in the figure below:
Another subsystem which has been modified is cpufreq. This traditional architecture for DVFS management in Linux uses an average sampling driven design that is prone to sub-optimal OPP selection. For example, a given choice of sampling rate may be too high or too low for a task with a very specific load profile. Making a wrong estimation of load can result in needlessly high OPP selection (wasted energy) or a low OPP selection (poor responsiveness).

The Linux scheduler community has long asked for scheduler driven DVFS management. The rationale has been that the scheduler is best positioned to estimate the load profile of a task that is to run and can therefore request the necessary amount of compute needed on a per-task basis. EAS introduces an implementation of the scheduler-driven DVFS technique known as sched-freq. The most important improvement that SchedFreq brings is that the scheduler can choose the frequency at which the CPU should run in the near future. This promotes more accurate frequency selection and therefore better servicing of the current load and utilisation.

The last module introduced by EAS is SchedTune, which uses the cgroups subsystem. SchedTune enables special case compute reservation for groups of tasks while also considering the energy impact. SchedTune is aimed for deployment in run-times with high visibility of compute requirements for tasks by way of a priori task classification. Middleware like Android fits very well in this category. Android knows which group of tasks require what compute for a given platform and at what point in time. SchedTune, by means of a cgroups exported interface, permits the Android run-time to efficiently move tasks between pre-created boost groups. SchedTune is aware of the platform specific energy model and

Figure 4 Load balance modifications and the check for an over-utilized system
works with the scheduler and SchedFreq to ensure that the relevant tasks are serviced as needed.

UI/UX intensive jobs are given special attention to meet their latency requirements) in the most energy efficient manner for a given platform.

## 2 Requirements

### 2.1 Platform

1. The platform must have a working cpufreq implementation to allow EAS to manage spare CPU capacity correctly.

2. The platform must have a working cpuidle implementation to take energy savings due to turning off relevant parts of the CPU topology into consideration.

3. To be able to create the energy model there has to be infrastructure, on the board (e.g. Juno's (ARM development platform) energy meter) or external (e.g. ARM energy probe) to measure energy consumption.

4. Non-overlapping power/performance curves for the core types (big.LITTLE architecture) increase potential energy savings which can be achieved with EAS.

### 2.2 Operating System

The current EAS implementation is targeted for a v3.18 Linux kernel based Linux or Android Operating System.
3 Source Code

The source code of the individual EAS components can be found here:

Linux Kernel:

- AOSP:
  
  https://android.googlesource.com/kernel/common/...

- Linaro LSK 3.18:

  https://git.linaro.org/landing-teams/working/arm/kernel-release.git/...

- Linaro LSK 4.4:

  https://git.linaro.org/landing-teams/working/arm/kernel-release.git/...

Userspace:

- Linaro Android Userspace:

  http://releases.linaro.org/android/reference-lcr/juno/7.0-16.10/

Tooling:

- Lisa Tool:

  https://github.com/ARM-software/lisa/releases/tag/v16.10

4 Related Documentation Sources

Additional information about the Energy Aware Scheduling can be found under:

https://developer.arm.com/open-source/energy-aware-scheduling
5 Functionality

5.1 Linux Kernel

EAS Configuration Data

Device Tree

This section describes the structure of the energy model that EAS relies on. Specifically, the section focuses on the use of the Flattened Device Tree (FDT) as a means of expressing the energy model. FDT is an established specification intended to describe platform properties in a hierarchical data structure. This data structure is expressed in a Device Tree Source (DTS) file. DTS files are compiled into binary blobs which are provided as inputs to the kernel. FDT bindings are specifications that describe methods to describe particular system properties. A special set of FDT bindings were created in order to describe energy models for EAS. This makes it possible to have a single kernel image which can be deployed on multiple platforms with different FDT blobs containing the appropriate energy model.

EAS relies on a simple platform energy cost model to guide scheduling decisions. The model only considers the CPU subsystem. The energy cost model concept is applicable to any system so long as the cost model data is provided for those processing elements in that system's topology that EAS is required to service.

Processing elements refer to hardware threads, CPUs and clusters of related CPUs in increasing order of hierarchy. At present, EAS only supports CPUs and clusters of CPUs. Only two clusters of CPUs are supported.

EAS requires two key cost metrics - busy costs and idle costs. Busy costs comprise of a list of compute capacities for the processing element in question and the corresponding power consumption at each capacity. Idle costs comprise of a list of power consumption values for each idle state [C-state] that the processing element supports.

These cost metrics are required for processing elements in all scheduling domain levels that EAS is required to service. Given that these cost metrics are properties of the system and have close topological ties to the system, it made sense to use the well-established Flattened Device Tree specification as a means to express these cost metrics to the kernel. For a complete description of the FDT bindings introduced for the cost model, please see the binding document located at Documentation/devicetree/bindings/scheduler/sched-energy-costs.txt within the EAS kernel source.

The best way to understand the structure of the energy model as described using FDT is to look at an example. A working knowledge of FDT is assumed. A snippet from the DTS file of a system composed of a cluster of 2 ARM Cortex-A57 CPUs and a cluster of 2 ARM Cortex-A53 CPUs is shown in the appendix subchapter “Example DTS file”.

EM data provisioning towards task scheduler

In the architecture specific topology shim layer the energy model data is constructed by calling init_sched_energy_costs(). Its interface towards the task scheduler is the scheduler domain topology level table (arm64_topology[]). It consists of struct sched_domain_topology_level entries each of which are extended by the function
pointer sched_domain_energy_f. This points the task scheduler to the scheduler domain specific energy data. The pointer is set to cpu_core_energy() for the MC scheduler domain level and to cpu_cluster_energy() for the DIE scheduler domain level.

Data structures

The struct sched_group_energy represents the per scheduler group related data which is needed for Energy Aware Scheduling. It contains:

- Number of elements of the idle state array
- Pointer to the idle state array which comprises 'power consumption' for each idle state
- Number of elements of the capacity state array
- Pointer to the capacity state array which comprises 'compute capacity and power consumption' tuples for each capacity state

The struct sched_group contains a pointer to a struct sched_group_energy data structure.

EAS Load Tracking

Overview

Per-Entity Load Tracking (PELT) implements load-tracking for the SCHED_NORMAL scheduling policy on per scheduling entity basis. Load stands for the actual load (a metric based on a sched-entity's runnable time, i.e. time spent on the run queue plus time spent running) and utilization (a metric based on a scheduling entity's running time). The fact that historical behaviour of a scheduling entity becomes increasingly less relevant with age when trying to draw conclusions about its future compute requirements is incorporated by using a geometric series for each of the two metrics. This has the natural effect of decreasing the relevance of the accounted time in each time slice as they age. PELT's bottom-up load-computation (scheduling entities contribute to the load of their parents’ (run-queues or task groups) allows the correct migration of load for an entity along with its accompanying entities. This provides the right metric for intelligent load-balancing especially when task groups are involved.

EAS introduces an additional PELT metric: PELT utilization. This metric is used for CPU frequency selection and wakeup task placement.

The Window Assisted Load Tracking scheme (WALT) offers an alternative load tracking mechanism to PELT. For mobile workloads there is evidence that using the window based approach of WALT for CPU frequency selection and wakeup task placement improves the performance/power ratio and responsiveness in comparison to the use of PELT.

FIE - Frequency Invariant Engine

The PELT implementation is prepared to be aware of frequency scaling to provide better estimates for cpu load and utilization by calling arch_scale_freq_capacity(). However the mainline kernel will only use the default implementation of arch_scale_freq_capacity() which always returns SCHED_CAPACITY_SCALE essentially not making PELT frequency scale-invariant.
Architectures interested in actual frequency scaling have to re-define arch_scale_freq_capacity() to point to their own frequency scaling solution.

The FIE is integrated into the cpufreq subsystem by scaling the per-cpu variable freq_scale with the current frequency of the cpufreq policy the CPU belongs to. Its interface function cpufreq_scale_freq_capacity() is used to provide the actual frequency-invariant scaling solution.

#define arch_scale_freq_capacity cpufreq_scale_freq_capacity

CIE - Cpu Invariant Engine

PELT is aware of CPU invariant scaling to provide better estimates for cpu utilization. CPU invariant scaling is necessary due to micro-architectural differences (i.e. instructions per cycles, for example) between CPUs and differences in the current maximum frequency supported by individual CPUs. However the mainline kernel will only use the default implementation of arch_scale_cpu_capacity() which always returns SCHED_CAPACITY_SCALE (1024) essentially not making PELT CPU scale-invariant.

The CIE is integrated into the architecture topology shim by scaling the per-cpu variable cpu_scale with the capacity value of the highest capacity state in the energy model of the CPU. Its interface function scale_cpu_capacity() is used to provide the actual CPU-invariant scaling solution.

#define arch_scale_cpu_capacity scale_cpu_capacity

WALT - Window Assisted Load Tracking

Window Assisted Load Tracking (WALT) provides a window based view of time in order to track the demand and CPU utilization of tasks. A "window" is a (compile time) configurable time duration, by default 20ms, which is used to collect a new "sample" for both task demand and CPU utilization. The start of a new window is synchronized across all the CPUs in the system. A "sample" measures:

- how long a task was RUNNING, within the corresponding window
- how long a CPU was BUSY, within the corresponding window

Samples are normalized to 1024, which represents the maximum utilization for both a task and a CPU. Thus, a 1024 sample means that a task was running for the whole duration of the corresponding window or, similarly, that a CPU was busy for the whole duration of the corresponding window.

Similar to PELT, WALT samples are also scaled such that they are architecture and frequency invariant. Architecture scaling compensates for the possibly different maximum capacity of CPUs while frequency scaling compensates for time spent running at different frequencies. Thus, for example a sample of value 512 is measured for a task running for the whole sample window duration but at a frequency which also provides half of the capacity of the CPU with the maximum capacity in the system.

The demand of a task is estimated by WALT considering the last N "non zero" collected samples, where N is a compile time configuration which is set to 5 by default. Thus, samples are collected for a task only for windows where the task had a chance to run. From all the collected samples the task utilization is estimated based on an "aggregation policy" which can be selected at compile time. By default the aggregation policy used is
WINDOW_STATS_MAX_RECENT_AVG which returns the maximum value between the average of all the collected samples and the most recent collected sample.

WALT estimates the utilization of a CPU by considering the sample measured during the last window. Thus, it’s noteworthy that everything happening in the current window’s time-frame is not affecting the view of CPU utilization.

Another main feature of WALT is that task demand and CPU utilization is tracked across all scheduling classes. The utilization metrics are associated with the struct task_struct for tasks and to the struct rq for cpus. The code to update these metrics is self-contained into a single source file (kernel/sched/walt.c) and functions exposed by this file are used only from the core scheduler code (kernel/sched/core.c). A custom set of “events” is defined which are used from specific call sites of the core scheduler to trigger updates of the metrics as well as to collect new samples.

Integration with EAS

It is possible to use WALT signals to drive EAS, for task placement as well as SchedFreq for OPP selection. The contact surface between WALT and EAS/SchedFreq is confined to a couple of functions: cpu_util() and task_util(). Both of these functions allow the transparent use of WALT signals instead of the corresponding PELT signal. When WALT is enabled at compile time via CONFIG_SCHED_WALT, a couple of control interfaces are exposed as procfs flags:

```
/proc/sys/kernel/sched_use_walt_cpu_util
/proc/sys/kernel/sched_use_walt_task_util
```

When set to 1, they force the previous functions to return the corresponding WALT signal.

NOTE: Although it’s technically possible to enable the usage of WALT only for one of the two signals, such mixed configurations have not been tested and they should be considered highly experimental.

Userspace exposed control interfaces

A set of configuration controllers are exposed to tune the behaviour of WALT:

- **sysctl_sched_walt_init_task_load_pct**
  
  The initial utilization (in percentage) of a newly created task.

- **sysctl_sched_walt_cpu_high_irqload**
  
  Time spent serving IRQs (in ms, by default 10) to consider a CPU highly loaded by interrupt. This value is used in find_best_target() to skip CPUs which are currently under a high IRQ pressure:

- **sysctl_sched_use_walt_cpu_util**
  
  Enable usage of WALT estimated CPU utilization.

- **sysctl_sched_use_walt_task_util**
  
  Enable the usage of WALT estimated task utilization.
Initial Task Load

Newly created tasks have no previous history for which a corresponding utilization can be estimated. A userspace control interface can thus be used to select the default initial task utilization for new tasks: `sysctl_sched_walt_init_task_load_pct`.

Since utilization influences both load balancing and clock frequency selection, choosing a particular value for this setting is a trade-off between potential performance gain experienced by the task in the early phases of its life and increased power consumption if tasks creation rate is too high.

Additional EAS related key concepts

Tipping Point - Over-Utilization

EAS is designed to save energy by managing spare capacity, i.e. CPU cycles, intelligently using a platform energy model. Without sufficient spare capacity, it is no longer feasible to optimize for energy and EAS optimizes solely for throughput instead. The tipping point at which EAS stops optimizing for energy is based on utilization.

A CPU is considered “over-utilized”, i.e. full, when its utilization leaves almost no capacity left:

\[
\text{capacity_of(cpu)} \times 1024 < \text{cpu_util(cpu)} \times \text{capacity_margin}
\]

`capacity_margin` is defined as 1280, i.e. 20%, by default. The entire system is considered over-utilized as soon as one CPU is over-utilized to ensure that no task is throughput constrained unnecessarily.

is_big_little

On a non big.LITTLE type of devices no simple heuristic exists to predict, before consulting the energy model, which cluster of CPUs is likely to be more energy efficient for wakeup load balancing decisions.

A tuning interface is exposed to user-space to be able to provide the scheduler such a high level topological information: `sched_is_big_little`.

EAS functionality

Energy aware wakeup path

When the system is not over-utilized, all tasks will occasionally wake up hence most of the energy-aware scheduling decisions can be made in the task wake-up path in the scheduler. If the system is over-utilized, tasks will be placed only on CPUs that are currently idle.

If `is_big_little` is true, tasks are generally placed according to the following steps:

1. The search space is narrowed down by selecting an appropriate high level `sched_group`, which is equivalent to a CPU cluster on most ARM systems. The selection is based on task utilization and per-CPU compute capacity.

2. A candidate CPU is chosen within the selected `sched_group` based on task utilization, CPU capacity, and current CPU utilization.
CPUs where the potential new utilization (current utilization plus task utilization) exceed the CPU capacity are discarded.

Any CPU that can accommodate the new utilization at the capacity offered by its current OPP is selected as a potential candidate. If it is already awake, search for further candidates is aborted.

A CPU with sufficient capacity at a higher OPP may be used as fall-back, if no other candidate is found.

3. If the previous CPU is over-utilized, the candidate is picked without further evaluation.

4. The choice between waking up the task on the previous CPU or the candidate is determined using the energy model, which evaluates the energy impact of the two options and the lowest impact option is chosen.

Wakeup path for a SchedTune boosted task on a non-big-little system

If sched_is_big_little is false and the system is not over-utilized tasks are generally placed according to the following steps (implemented by the find_best_target() function):

1. It is established if the task for which a decision has to be taken is boosted or not (being SchedTune boosted affects several parts of the selection heuristic).

2. It is established if the task for which a decision has to be taken prefers idle CPUs (prefer_idle is true) or not (as this affects several parts of the selection heuristic).

3. All the CPUs available on the system and that meet the task affinity settings are iteratively considered:

   - starting from the first CPU for non-boosted tasks

   - starting from the CPU with the highest id for boosted tasks (assuming that CPUs with higher ids can deliver higher peak performance)

4. A candidate CPU is chosen based on task (boosted) utilization, prefer_idle settings, CPU capacity and current CPU utilization:

   - CPUs where the potential new utilization (current utilization plus task (boosted) utilization) exceed the CPU capacity are discarded.

   - The first CPU that is found idle for a prefer_idle task is annotated, and then unconditionally selected as candidate to improve latency.

   - CPUs with sufficient capacity at a higher OPP are considered as backup (only one CPU following a least capacity policy is finally annotated) in case a CPU with enough capacity at current OPP is not eventually found.

   - Any CPU that can accommodate the new utilization at the capacity offered by its current OPP is selected as a potential candidate:

     - following a lowest utilization policy for prefer_idle tasks

     - following a highest utilization policy for non prefer_idle tasks

   If the task is boosted or if it prefers idle CPUs, and the selected CPU is idle, the candidate CPU is returned without consulting the energy model.
Misfit Task

Tasks that run for longer periods don't regularly come through the wake-up path and therefore don't get a chance of being placed on higher capacity CPUs through that route. These tasks have to be migrated as part of periodic load-balancing or idle load-balancing instead. CPUs with runnable tasks with a higher utilization than can be accommodated mark themselves as having one or more “misfit” tasks. In this scenario, the system is over-utilized and scheduling decisions consider only throughput.

During periodic or idle load-balancing, if no general overload issues are present, CPUs with misfit tasks are considered and misfit tasks are migrated to more appropriate CPUs if necessary.

SchedFreq - Scheduler driven DVFS

SchedFreq is an EAS module that makes it possible to control CPU clock frequency selection directly from the scheduler. SchedFreq works as a shim layer between the scheduler and the CPUFreq framework. This enables the scheduler to implement a CPUFreq policy governor by itself, basically replacing legacy governors such as ondemand, interactive, etc. The tight connection between the scheduler and clock frequency selection provides better system-wide policies and improves both performance and power savings.

SchedFreq can be activated through the usual CPUFreq sysfs interfaces.

Up and down threshold configuration interfaces

SchedFreq exposes to user-space an up and a down throttling threshold. Throttling thresholds (beyond the physical transition latency limit) are necessary to prevent too frequent (and potentially harmful) clock frequency transitions. In general it is required to be quick in responding to a sudden increase in utilization, and have a bit of hysteresis for brief drops. The actual values (in nanoseconds) can be configured via two governor sysfs files: up_throttle_nsec and down_throttle_nsec.
SchedTune - Task classification and control

SchedTune is an EAS module that implements a single (and simple) central tunable controller to balance energy-efficiency and performance-boosting. It extends the SchedFreq CPUFreq governor, to bias the OPP selection, thus allowing this governor to provide behaviours similar to other governors; e.g. interactive, performance or powersave. It also integrates the energy-aware wakeup path of the EAS core, to bias tasks placement, thus allowing to trade-off energy-efficiency for performance-boosting.

From a user-space perspective, SchedTune fosters the collection of sensible information to better support the scheduler in its decisions. A simple yet powerful interface, based on the cgroups API, allows an easy integration with run-times available in platforms like Android. This interface is suitable to support task classifications, for example foreground vs background, which can then be managed according to different goals from the scheduling standpoint.

SchedTune is exposed to user-space via the cgroups interface, where the schedtune controller can be mounted to get access to the sysfs tunables it exposes. For example, Android mounts it by default under /dev/stune. This path represents the root boostgroup, which includes every task in the system. Additional boostgroups can be created (up to a maximum of 5 with the default kernel configuration), to represent different possible tasks classifications. All tasks within the same boostgroup are assigned a similar set of SchedTune parameters.
Userspace exposed control interfaces

For each boostgroup, SchedTune currently exposes two tunables:

1. schedtune.boost (int [-100..100])

This parameter specifies the boost percentage value which is used for the tasks in this group.

This value is used to compute a "margin" to be added or removed to or from the utilization signal of a task/cpu. The value of the margin is calculated to provide a well-defined and expected user-space behaviour. For example, the following table reports the meaning of some specific boost values:

<table>
<thead>
<tr>
<th>Boost value [%]</th>
<th>Meaning (e.g. run the task at a frequency corresponding to)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Minimum required capacity (max energy efficiency)</td>
</tr>
<tr>
<td>100</td>
<td>Maximum possible speed (min time to completion) (*)</td>
</tr>
<tr>
<td>50</td>
<td>Something in between the previous two configurations</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>-50</td>
<td>Half of the minimum required capacity</td>
</tr>
<tr>
<td>-100</td>
<td>Minimum available capacity (minimum OPP)</td>
</tr>
</tbody>
</table>

(*) minimum latency is not yet completely supported in the current ACK release, this feature is a work in progress and will be added in a following release.

The logic to convert the boost value into a proper margin is based on a "Signal Proportional Compensation" (SPC) policy which is implemented in:

```
kernel/sched/fair.c::schedtune_margin(signal, boost)
```

The parameter `signal` is the original task or CPU utilization to compensate and `boost` is the boosting percentage defined for the boostgroup the task is part of.

2. `schedtune.prefer_idle (int [0,1])`

This parameter specifies the "packing" vs "spreading" strategy to be used for tasks placement in the wake up path.

One of the first operations done in the energy aware wakeup path is the selection of a candidate CPU where the task should be executed. The `prefer_idle` flag reflects the desire for low-latency activation, possibly at the expense of increased energy consumption. Tasks belonging to boostgroups having this flag set are allocated (if available) on an idle CPU, thus reducing to the minimum their activation latency. When the flag is reset (or not IDLE CPUs are available) a more complex CPU selection heuristic is used, which targets tasks packing to optimize for energy consumptions.

The figure below shows the state of the kernel scheduler and SchedTune in time t0, t1. It narrows down the case when the boost is applied via SchedTune to the run queue utilization. Thanks to this enhancement, it is possible to request more capacity via SchedFreq.
Dynamic CPU capacity capping

The current maximum frequency of a core in a multi-core processor can be restricted to a value lower than its absolute maximum frequency. One of the reasons for this to happen is thermal management. It makes sure that the system always operates in the boundary of its power budget. This is normally achieved by reducing the maximum frequency of CPU cores. To make sure that the task scheduler knows about this new maximum frequency dynamic CPU capacity capping can be applied.

Dynamic CPU capacity capping provides PELT and the scheduler CPU capacity management with a maximum frequency scaling correction factor. This scaling factor describes the influence of running a CPU with a current maximum frequency lower than the absolute maximum frequency.

\[
current\_max\_freq(cpu) \ll \text{SCHED\_CAPACITY\_SHIFT} / max\_freq(cpu)
\]

Dynamic CPU capacity capping is integrated into the cpufreq subsystem by scaling the per-cpu variable \text{max\_freq\_scale} with the current maximum frequency of the cpufreq policy the CPU belongs to. Its interface function \text{cpufreq\_scale\_max\_freq\_capacity()} is used in the CIE implementation making sure that Dynamic CPU capacity capping affects PELT and CPU capacity management.

\[
@t0: P0 and P1 are runnable:
\begin{align*}
\text{util\_avg} &= 120, \text{max\_boost} = 0 \\
\text{boost\_cpu\_util}(120, 0) &= 120
\end{align*}
\]

\[
@t1: P0 and P2 are runnable:
\begin{align*}
\text{util\_avg} &= 140, \text{max\_boost} &= 50 \\
\text{boost\_cpu\_util}(140, 50) &= 140 + (1024 – 140) \times 50\% = 582
\end{align*}
\]
EAS Trace

The EAS release contains several patches with additional trace events which help in verifying that the individual EAS building blocks such as load-tracking (PELT and WALT), FIE, CIE, SchedFreq and SchedTune are working correctly. These trace events are also required for the Interactive Test and Analysis part of the LISA toolkit to work correctly.

EAS Debug

There is support to evaluate the EAS energy model on a running system via the proc file-system. In case CONFIG_SCHED_DEBUG and CONFIG_SYSCTL are enabled the related files are placed under the subdirectory named energy inside the /proc/sys/kernel/sched_domain/cpuX/domainY/groupZ directory.

To be able to figure out the reason for a potential misbehaviour of a task while running the EAS verification tests the cause of each task migration is additionally provided with the existing sched_migrate_task trace event. Please refer to chapter “Evaluate per task behavior” for more information.
5.2 User-space

It is a fact that a policy to minimise energy use is not always suitable for some tasks in an interactive system. This is not a problem caused by EAS but a feature of any system where performance is constrained to reduce energy consumption. It takes some time to recognise a task as having a high compute requirement. Similarly, there are some tasks for which minimum latency is more important than minimum energy to complete a job. Where you have heterogeneous multiprocessing, you also have to select a suitable CPU match for a task.

There have been attempts at controlling this behaviour for at least as long as systems have implemented DVFS - the interactive governor is a longstanding Android feature which links to user input to try and minimise latency. Some systems use cpusets to guide tasks to specific CPUs and they may use cpu bandwidth controls and/or task priority to control behaviour. There are also controls which modify scheduler parameters and hence change the frequency of load balance decisions etc. Usually a combination of all these tools is used to configure a system to provide an acceptable balance between energy use and performance. The old-style (HMP or GTS) big.LITTLE MP support had some userspace controllable tuning parameters and it was not uncommon to see userspace daemons dynamically controlling those thresholds in response to specific system events or behaviours. All of these tools and techniques are a way to guide the kernel as to the desired response when scheduling.

Part of the EAS solution described above provides a cgroup controller to allow tasks to be classified and different scheduler algorithms to be used for those task classes. Future versions of the AOSP middleware will be making use of this new controller to control interactive response for some key tasks.

This document will be updated to describe that implementation and what is needed to make use of it on another platform when that implementation is published. In addition, ARM will be providing an implementation of the required parts to enable this functionality for Android on the Juno platform through Linaro Android.

Android 7.1 AOSP changes for Google Pixel

The EAS/stune userspace integration, done as part of the Pixel program, has been made available in AOSP android_7.1.0_r1. These will make their way into Linaro Android when this release has been taken in. All integration points mentioned here are for the tag ‘android_7.1.0_r1’. Linaro LCR Android release 16.10 is based on android_7.0 and has only basic support for foreground/background stune groups.

SchedTune CGroup layout

There are 4 groups defined for schedtune, commonly abbreviated to stune in AOSP commit messages.

<table>
<thead>
<tr>
<th>Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/stune</td>
<td>root group: anything not explicitly placed elsewhere goes here</td>
</tr>
<tr>
<td>/dev/stune/background</td>
<td>background group: this group holds tasks which the</td>
</tr>
</tbody>
</table>
userspace designates as unimportant for interactive performance

<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
<th>Includes</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/stune/foreground</td>
<td>foreground group, this group holds tasks which the userspace designates as important for interactive performance</td>
<td>audioserver / mediaserver</td>
</tr>
<tr>
<td>[prefer_idle=1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/dev/stune/top-app</td>
<td>top-app: holds tasks which belong to the application top-most of the display stack, which use SP_TOP_APP scheduling policy</td>
<td>cameraserver / bootanimation</td>
</tr>
<tr>
<td>[prefer_idle=1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A thread group held with SP_TOP_APP which is to be set using set_process_policy and should be treated as a foreground thread.</td>
<td></td>
</tr>
<tr>
<td>[boost=10]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SchedTune CGroup usage for Services**

Some system components have been placed directly into one of the stune groups. The init script sections for these services are updated to write the service PID into the relevant group at creation time.

1. /dev/stune/foreground
   a. audioserver
   b. mediaserver
2. /dev/stune/top-app
   a. cameraserver
   b. bootanimation

**SchedTune CGroup usage for Application Tasks**

All tasks in Android are mapped to specific scheduler policies by platform/system/core/libutils/sched_policy.c. This includes assigning to specific cpuset groups as well as now assigning to the different schedtune groups. In order that sched_policy.c will include the schedtune group mappings, you must have USE_SCHEDBOOST defined when building. ActivityManager decides which tasks belong to a foreground app, a background app, or the top-app and passes the policy down through setProcessGroup etc. until eventually sched_policy.c writes the TIDs into the correct tasks file.

Further to this, each application has an android.ui thread which will have its thread group set to TOP_APP. This results in calling into sched_policy.c:set_sched_policy and should force that task to be added to the foreground cpuset group and the top-app schedtune group.
AOSP Commits adding EAS Integration

- [https://android.googlesource.com/platform/system/core/+/11cde56](https://android.googlesource.com/platform/system/core/+/11cde56) Mount schedtune cgroup as /dev/stune

- [https://android.googlesource.com/platform/system/core/+/770ee49](https://android.googlesource.com/platform/system/core/+/770ee49) & [https://android.googlesource.com/platform/system/core/+/aa45cb8](https://android.googlesource.com/platform/system/core/+/aa45cb8) Set stune groups from set_sched_policy

- [https://android.googlesource.com/platform/system/core/+/5dcff8f](https://android.googlesource.com/platform/system/core/+/5dcff8f) Add support for background stune group

- [https://android.googlesource.com/platform/system/core/+/481edfe](https://android.googlesource.com/platform/system/core/+/481edfe) & [https://android.googlesource.com/platform/system/core/+/955694b](https://android.googlesource.com/platform/system/core/+/955694b) Add support for the top-app stune group

- [https://android.googlesource.com/platform/frameworks/av/+/64c1ce8](https://android.googlesource.com/platform/frameworks/av/+/64c1ce8) & [https://android.googlesource.com/platform/frameworks/av/+/caba519](https://android.googlesource.com/platform/frameworks/av/+/caba519) Put mediaserver and audioserver in the foreground stune group.

- [https://android.googlesource.com/platform/frameworks/av/+/052c495](https://android.googlesource.com/platform/frameworks/av/+/052c495) Put cameraserver in the top-app stune group.

- [https://android.googlesource.com/platform/frameworks/base/+/1c14fbc](https://android.googlesource.com/platform/frameworks/base/+/1c14fbc) &

- [https://android.googlesource.com/platform/frameworks/base/+/5c52691](https://android.googlesource.com/platform/frameworks/base/+/5c52691) Put bootanimation in the top-app stune group.

- [https://android.googlesource.com/platform/frameworks/base/+/fe51b8f](https://android.googlesource.com/platform/frameworks/base/+/fe51b8f) & [https://android.googlesource.com/platform/frameworks/base/+/4074ad0](https://android.googlesource.com/platform/frameworks/base/+/4074ad0) Set android.ui to be in the fg stune group. (but uses top-app, so with later patches results in android.ui being in the top-app stune & fg cpuset group).

Userspace Integration required from OEMs

You will notice that there is no ‘interactive’ style responsiveness boosting implemented in the kernel for the sched-freq CPUFreq governor. If you want this functionality on your device, then the place to implement this is in the PowerHAL - you can set minimum frequencies, change stune group boost levels and more in response to indications from Android about what activities are going on. We do not discuss the implementation of a PowerHAL, but there is a Hikey-specific PowerHAL available in the Hikey device project for AOSP. Since the Android 7.1 release in AOSP, the new hints for Android 7.1 are implemented in a Pixel-specific PowerHAL.

When looking at PowerHAL implementations for other devices please remember that the actions taken on receipt of hints from the middleware are normally device-specific tunings and should be carefully tested.

New Android 7.1 Power Hints

For more detail, refer to your Android PDK. For reference, the new hints are named like so:

- POWER_HINT_SUSTAINED_PERFORMANCE
- POWER_HINT_VR_MODE
6 Tuning

This chapter contains the description of task scheduler related configuration data which can be used to fine tune the EAS behaviour on a particular target device.

- sched_migration_cost

  This value describes the amount of time after the last execution of a task that this task is considered to be "cache hot" in load balancing decisions. The default value is 500000 ns.

  A "cache hot" task is not considered for periodic, idle or nohz-idle load balance as long as the scheduler domain sd->nr_balance_failed counter is smaller or equal then the sd->cache_nice_tries value (leave "cache hot" task for # tries on the current cpu) plus 2.

  If the idle time of a certain cpu or cluster is higher than desired when there are runnable tasks in the system, try to reduce this value.

  This task scheduler sysctl tuning variable is located in /proc/sys/kernel/.

- busy_factor

  In case a cpu is busy, i.e. the cpu is not idle, the load balancing operation interval is increased by multiplying the scheduler domain sd->balance_interval value with this factor.

  The time between consecutive load balancing procedures on a specific scheduler domain level of a busy cpu can be reduced by decreasing this value.

  This scheduler domain data is set to 32 during sched domain hierarchy bring-up in sd_init().
7 Integration

7.1 An Overview of the LISA Toolkit

The Linux Interactive System Analysis (LISA) is a toolkit containing libraries and APIs that are suitable for interactive analysis of the behaviour of a Linux based system. The main goals of this toolkit are:

1. To support studying and understanding of existing behaviours, e.g. how PELT works
2. To support the analyses of new features to verify their impact on existing behaviours
3. To get possible insights on what's not working or not working well and possibly why
4. To establish a common framework and language to share easy to understand and reproducible experiments

This toolkit is targeted at two main consumers: developers and integrators.

Developers are people involved in the creation of new features of a Linux based system, mainly on the kernel-space side but user-space developers may also be involved. Their goal is usually to run some experiment on a target platform, collect execution traces and use the collected data to generate plots and statistics which allow the comparison of existing behaviours with the new features that are being developed. This process is usually repeated multiple times during the development and testing of a new feature and involves the interaction of possibly multiple different target platforms.

Integrators are people mainly interested in running regression tests and performing analyses to verify how the behaviours of a system are changing across different release versions.

The working flow used by both of these customers is usually repeated multiple times and it may involve interactions with multiple different target platforms. Moreover, developers usually need to share their results and experiments with other developers in order to reproduce the same experiments or to cross-check conclusions.

The LISA toolkit addresses all these requirements and promotes re-use and cooperation among different people using different targets. LISA is a collection of python libraries which expose different sets of APIs at different abstraction levels as shown below:
Figure 8 Bird’s eye view on the LISA toolkit components

Hardware Abstraction Layer

The devlib library is used to abstract the access to a target device. A device can be either a remote machine, accessed via SSH or ADB, or the local one. The devlib library allows the configuration of an abstract communication channel to send commands to the device as well as exchange files with the device. On top of these basic APIs, devlib also provides a set of convenient libraries to access and control the main subsystems of a Linux based device, e.g.: cpufreq, cpuidle, hotplug, cgroups, ftrace, etc.

An example of the devlib APIs is available at this link:
http://github.com/ARM-software/lisa/blob/master/ipynb/tutorial/02_TestEnvUsage.ipynb

Data Collection and Analysis

A set of libraries are provided for processing collected trace data and for analysing system behaviours.

The TRAPPy library (TRace Analysis and Plotting in Python) provides support for parsing a trace of system events, collected using either ftrace or systrace and converting them into PANDAS dataframes. This conversion gives access to rich PANDAS APIs for analysis of the collected data.

An example of the TRAPPy APIs is available at this link:
http://github.com/ARM-software/lisa/blob/master/ipynb/tutorial/05_TrappyUsage.ipynb

The BART library (Behavioural Analysis and Regression Toolkit) provides support for analysing system behaviours and assertion based testing for specific behaviours of interest. The API exposed by this library is mainly intended for the development of regression tests.

An example the BART APIs is available in the “Behavioural Analysis” section of this link:
https://github.com/ARM-software/lisa/blob/master/ipynb/tutorial/UseCaseExamples_SchedTuneAnalysis.ipynb

Interactive Test and Analysis

One of the main advantages of LISA is the ability to define a convenient workflow to define new tests and experiments, execute them on multiple targets, process the collected data and share all of these with other developers. This greatly simplifies the reproduction and verification of the results. This flow is based on (but not limited to) the usage of IPython notebooks as the primary environment where experiments are coded and results produced. An IPython notebook is just an interactive Python shell, running within a browser, which makes it easy to mix code which produces results and additional comments, thus making it a suitable environment to support the "interactive analysis" which LISA is fostering. Notebooks are not only a playground to design and run experiments but they are also a native exchange format which can be both used to share reproducible results or simply to publish an analysis report.

An example usage of the LISA's APIs to perform an interactive analysis is available at this link:

https://github.com/ARM-software/lisa/blob/master/ipynb/tutorial/UseCaseExamples_SchedTuneAnalysis.ipynb

Automated Tests

If needed, notebooks can be easily converted into standalone tests. For this purpose, LISA provides an API which allows the user to encapsulate the code of a notebook into a standard python nosetest class. These tests are useful for example to run regression analyses across different versions and releases of a code base.

7.2 Integrating a board into LISA

A few steps are required to integrate a new board with LISA. First of all, we have to distinguish between what we call a platform and a board in the LISA configuration. A board is the target device intended for analysis; whereas the platform identifies the type of system running on that device. For example, if Linux is running on a Juno development board that is to be experimented with, from the point of view of the LISA configuration, the board is Juno and the platform is Linux.

That said, in order to integrate a board, LISA requires the following information in a JSON file:

A board description containing:

- A list of CPU names
- The name of the big core in the system
- An energy model that can be generated later as described in the following section

Refer to the following link for more details on the format of the JSON file:

Note that currently LISA only supports devices with up to two clusters while designs with more than two clusters are only partially supported (some functionality may not be available).

For energy and power measurements, LISA supports three different energy meters: Linux Hardware Monitors (HWMON), ARM Energy Probe (AEP), BayLibre ACME Cape (ACME). For integrating a new energy meter, please refer to the following Wiki page:


7.3 Building an Energy Model

LISA provides an energy model building feature for platforms where energy can be measured either at cluster level or at system level (for example, battery energy or the CPU's nearest measurement point).

In order to use this feature, the following requirements must be satisfied:

5. The desired energy probe must be integrated with LISA

6. For each cluster that is intended to be profiled, the following information must be provided:
   - cluster name
   - energy probe channel name
   - name of the core in the cluster
   - list of CPU IDs belonging to the cluster
   - list of frequencies available in the cluster
   - list of idle states available in the cluster

7. A mapping between cluster names and cluster IDs used in the kernel must be provided

For a more detailed description of how to use this feature as well as reference examples, please have a look at the notebooks section available in LISA:

https://github.com/ARM-software/lisa/tree/master/ipynb/energy

7.4 Verifying EAS functionality

EAS acceptance tests

The EAS acceptance tests are a set of four test cases written using BART that verifies four behaviours that we expect EAS to follow.

These tests are part of lisa. To run them, configure your target in target.config and run:

nositests -v tests/eas/acceptance.py
Fork Migration

Goal
Check that newly created threads start on a big CPU.

Detailed description
The test spawns as many threads as there are cores in the system. It then checks that all threads started on a big core.

Expected behaviour
The threads start on a big core.

Small Task Packing

Goal
Many small tasks are packed in little cpus.

Detailed description
The tests spawn as many tasks as there are cpus in the system. The tasks are small, so none of them should be run on big cpus and the scheduler should pack them on little cpus.

Expected behaviour
All tasks run on little cpus.

Offload migration and idle pull

Goal
Big cpus pull big tasks from little cpus when they become idle.

Detailed description
This test runs twice as many tasks as there are big cpus. All these tasks are big tasks. Half of them are called "early_starter" and the other half "migrator". The migrator tasks start 1 second after the early_starter tasks. As the big cpus are fully utilized when the migrator tasks start, some tasks are offloaded to the little cpus. As the big cpus finish their tasks, they pull tasks from the little to complete them.

Expected behaviour
As there are as many early_starter tasks as there are big cpus, the early_starter tasks should run in the big cpus until they finish. When the migrator tasks start, there is no spare capacity in the big cpus so they run on the little cpus. Once the big cpus finish with the early_starters, they should pull the migrator tasks and run them.

It is possible that when the migrator tasks start they do it on big cpus and they end up displacing the early starters. This is acceptable behaviour. As long as big cpus are fully utilized running big tasks, the scheduler is doing a good job.

That is why this test doesn't test for migrations of the migrator tasks to the bigs when we expect that the early starters have finished. Instead, it tests that:

- The big cpus are fully loaded as long as there are tasks left to run in the system
- The little cpus run tasks while the big cpus are busy (offload migration)
- All tasks get a chance to run on a big cpu (either because they started there or because of idle pull)
Wake migration

Goal

A task that switches between being big and little cores moves to big and little cores accordingly.

Detailed description

This test creates as many tasks as there are big cpus. The tasks alternate between high and low utilization. They start being small load for 5 seconds and they become big for another 5 seconds, then small for another 5 seconds and finally big for the last 5 seconds.

Expected behaviour

The tasks should run on the little cpus when they are small and in the big cpus when they are big.

Capacity capping testing

Similar to the acceptance tests, this test is part of lisa. To run it configure your target in target.config and run:

nose tests -v tests/eas/capacity_capping.py

Goal

Verify that dynamic CPU capacity capping works in the system.

Detailed Description

The maximum frequency of a core can be restricted to a lower value than its absolute maximum frequency. This may happen because of thermal management or as a request from userspace via sysfs. Dynamic CPU capacity capping provides PELT and the scheduler CPU capacity management with a maximum frequency scaling corrector which describes the influence of running a CPU with a current maximum frequency lower than the absolute maximum frequency.

The test creates as many busy threads as there are big cpus. These busy threads have high load and should run in the CPUs with highest capacity. The test has three phases of equal length. In the first phase, the system runs unconstrained. In the second phase, the maximum frequency of the big cpus is limited to the lowest frequency that the big frequency domain can run at. Finally, in the third phase, the maximum frequency of the big cpus is restored to its absolute maximum, i.e. the system is unconstrained again.

This test assumes that the lowest OPPs of the big cpus have less capacity than the highest OPP of the little cpus. If that is not the case, this test will fail. Arguably, capacity capping is not needed in such a system.

Expected Behaviour

The threads have high load, so they should always run in the CPUs with the highest capacity of the system. In the first phase the system is unconstrained, so they should run on the big CPUs. In the second phase, the big cluster’s maximum frequency is limited and the little CPUs have higher capacity. Therefore, in the second phase of the test, the threads should migrate to the little cpus. In the third phase the maximum frequency of the big cpus is restored, so they become again the
CPUs with the highest capacity in the system. The busy threads must run on the big cpus in the third phase.

**Analyze test failures**

Fully debugging why a test has failed is outside the scope of this document. However, the first step is usually to plot the trace and have a look at the scheduling decisions. We can do this with trappy using an ipython notebook. For example:

```python
import trappy

trace_file = "../results_latest/offload_idle_pull.dat"

trappy.plotter.plot_trace(trace_file)
```

![Figure 9 Lisa example plot](image)

The trace files for the different tests are:

- Fork Migration: fork_migration.dat
- Small Task Packing: small_task_packing.dat
- Offload migration and idle pull: offload_idle_pull.dat
- Wake Migration: wake_migration.dat
Evaluate per task behavior

During EAS integration on a new platform a test case might fail due to misbehaviour of one of the test tasks. This misbehaviour is normally characterized by an unexpected task migration during the test run, i.e. there are occurrences in which the test task is not scheduled on the designated cpu core type.

To be able to analyse during trace file post processing which scheduler core path (e.g. EAS wakeup or periodic load balance) led to this misbehaviour CONFIG_SCHED_DEBUG_EAS_MIGRATION can be enabled so that the migration cause is additionally provided with each sched_migrate_task trace event:

ksoftirqd/1-17 [001] 49.672428: sched_migrate_task: comm=usb-storage pid=1048 prio=120 orig_cpu=1 dest_cpu=2
cause=select_idle_sibling:idle_cpu

ksoftirqd/1-17 [001] 49.685420: sched_migrate_task: comm=bash pid=2074 prio=120 orig_cpu=1 dest_cpu=2
cause=select_idle_sibling:default

hackbench-2074 [002] 49.697317: sched_migrate_task:
comm=hackbench pid=2112 prio=120 orig_cpu=5 dest_cpu=2
cause=select_task_rq_fair:find_idlest_group/cpu

The set of individual task migration causes is defined in include/trace/events/sched.h.

You can visualize the events by using trappy in an ipython notebook. For example, if we want to analyse the result of the wake migration test we would do:

import trappy

trace_file = "../results_latest/wake_migration.dat"

trace = trappy.FTrace(trace_file)

trappy.plotter.plot_trace(trace, execnames=["wmig", "wmig1"])

This plot shows the scheduling decisions for the two tasks that comprise the test in this platform: wmig and wmig1. If we want to know the reason behind, for example, the migration in cycle 6.039334, we can do so by showing the events that happened around that time with this code:

trace.sched_migrate_task.data_frame[5.9:6.15]
In [1]:
```python
import trappy
trace_file = "./results/latest/wake_migration.dat"
trace = trappy.Trace(trace_file)
trappy.plotter.plot_trace(trace, execnames=['wmig', 'wmigl'])
```

```
A = 6.149680
```

Figure 10 Ipython notebook example plot

```ini
[5.9:6.15]
```

```
<table>
<thead>
<tr>
<th></th>
<th>_comm</th>
<th>_cpu</th>
<th>_pid</th>
<th>cause</th>
<th>comm</th>
<th>dest_cpu</th>
<th>orig_cpu</th>
<th>pid</th>
<th>prio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
<td></td>
<td></td>
<td>migration/5</td>
<td>5</td>
<td>27</td>
<td>select_idle_sibling/idle-cpu</td>
<td></td>
<td>1713</td>
</tr>
</tbody>
</table>
```

Figure 11 Analysis within kernelshark

Copyright © 2016 ARM Limited or its affiliates. All rights reserved.
7.5 EAS test coverage

This software release has been fully tested on a dual-cluster SMP platform with different maximum cpu frequency and different OPPs for the cpu cores on both clusters.

Additional testing was performed on a Juno ARM development platform (big.LITTLE architecture) as well as on a HiKey board, which is a dual-cluster SMP platform with no cpu frequency differences between the two clusters and a frequency domain spanning all cpus in the system.


8 Glossary

Scheduling Entity

A scheduling entity (struct sched_entity) describes the unit which can be scheduled by the task scheduler. It can represent a task as well as a task group.

Task Group

A task group (struct task_struct) is an abstraction for a group of tasks which is represented by a single scheduling entity. The scheduler normally operates on tasks but in certain configurations it may be desirable to group tasks and provide fair CPU time to each such task group rather than to each individual task.

Scheduling Domain

A scheduling domain (struct sched_domain) is a set of CPUs which share properties and scheduling policies and which can be balanced against each other. Scheduling domains are hierarchical. A multi-level system will have multiple levels of domains.

E.g. the multi-cluster level (MC level) contains all the cpus belonging to a certain cluster whereas the physical processor level (DIE level) spans all the cpus of the processor.

Scheduling Group

Each scheduling domain contains two or more scheduling groups (struct sched_group) which are treated as a single unit by the scheduling domain. When the scheduler tries to balance the load within a scheduling domain, it tries to even out the load carried by each scheduling group without worrying directly about what is happening within the scheduling group.

Frequency Invariance

Frequency invariance makes the load and utilization signal of Per-Entity Load-Tracking (PELT) aware of CPU frequency scaling.

Without frequency invariance a task with 25% load on a CPU operating at 100% of its maximum frequency would change its load to 50% in case the frequency decreases to 50% of the maximum frequency. With frequency invariance the load of the task remains 25% regardless of the CPU frequency. The same is true for the utilization signal.

CPU Invariance

CPU invariance makes the utilization signal of Per-Entity Load-Tracking (PELT) and task scheduler CPU capacity management aware of CPU micro-architectural differences and/or differences in the maximum frequency supported by the CPUs.

Without CPU invariance a task with 25% utilization on a CPU with a capacity of 100% of the system-wide maximum CPU capacity would change its utilization to 50% in case it migrates to a CPU with a capacity of 50% of the system-wide maximum CPU capacity. With CPU Invariance the utilization of the task remains 25% regardless of the CPU the task is running on.

Utilization


The utilization of a scheduling entity is the amount of time the scheduling entity is running on a CPU over an elapsed period of time.

Load

The load of a scheduling entity is the amount of time the scheduling entity is ready to run on a CPU (i.e. it is runnable) multiplied by its weight (e.g. the weight of a task is its priority) over an elapsed period of time.

Spreading

The goal of the CFS task scheduler on an SMP platform is to distribute (hence to spread) work evenly across all available CPUs to guarantee maximum performance and minimum latency. This behaviour is characteristic for a work-conserving scheduler which tries to keep all scheduled resources busy as long as there are scheduling entities ready to be scheduled.

Packing

The goal of the EAS enhancement of the CFS task scheduler on an SMP platform is to maximize the power efficiency without harming the overall system throughput. EAS tries to distribute scheduling entities on the smallest number of suitable CPUs (hence to pack) while still meeting their compute requirements. This allows the power management subsystems to potentially save energy by turning off unused parts of the processor.
9 Appendix

9.1 Example DTS file

cpus {
    #address-cells = <2>;
    #size-cells = <0>;
    
    A57_0: cpu@0 {
        compatible = "arm,cortex-a57","arm,armv8";
        reg = <0x0 0x0>;
        device_type = "cpu";
        enable-method = "psci";
        next-level-cache = <&A57_L2>;
        clocks = <&scpi_dvfs 0>;
        cpu-idle-states = <&CPU_SLEEP_0 &CLUSTER_SLEEP_0>;
        sched-energy-costs = <&CPU_COST_0 &CLUSTER_COST_0>;
    };
    
    A57_1: cpu@1 {
        compatible = "arm,cortex-a57","arm,armv8";
        reg = <0x0 0x1>;
        device_type = "cpu";
        enable-method = "psci";
        next-level-cache = <&A57_L2>;
        clocks = <&scpi_dvfs 0>;
        cpu-idle-states = <&CPU_SLEEP_0 &CLUSTER_SLEEP_0>;
        sched-energy-costs = <&CPU_COST_0 &CLUSTER_COST_0>;
    };
};
A53_0: cpu@100 {
    compatible = "arm,cortex-a53","arm,armv8";
    reg = <0x0 0x100>;
    device_type = "cpu";
    enable-method = "psci";
    next-level-cache = <&A53_L2>;
    clocks = <&scpi_dvfs 1>;
    cpu-idle-states = <&CPU_SLEEP_0 &CLUSTER_SLEEP_0>;
    sched-energy-costs = <&CPU_COST_1 &CLUSTER_COST_1>;
};

A53_1: cpu@101 {
    compatible = "arm,cortex-a53","arm,armv8";
    reg = <0x0 0x101>;
    device_type = "cpu";
    enable-method = "psci";
    next-level-cache = <&A53_L2>;
    clocks = <&scpi_dvfs 1>;
    cpu-idle-states = <&CPU_SLEEP_0 &CLUSTER_SLEEP_0>;
    sched-energy-costs = <&CPU_COST_1 &CLUSTER_COST_1>;
};

energy-costs {
    CPU_COST_0: core-cost0 {
        busy-cost-data = <
            417  168
            579  251
            744  359
            883  479
            1024 616
        >;
    },
idle-cost-data = <
  15
  0
>;
}

CPU_COST_1: core-cost1 {
  busy-cost-data = <
    235 33
    302 46
    368 61
    406 76
    447 93
  >;
  idle-cost-data = <
    6
    0
>;
}

CLUSTER_COST_0: cluster-cost0 {
  busy-cost-data = <
    417 24
    579 32
    744 43
    883 49
    1024 64
  >;
  idle-cost-data = <
    65
    24
>;
The first CPU node in the example above labelled A57_0: cpu0 represents a Cortex-A57 CPU and follows standard conventions as described in the traditional CPU bindings document. A new property called sched-energy-costs is introduced which is a list of phandles to cost nodes. The order of phandles in the list is significant. The first phandle is to the current processing element's own cost node. Subsequent phandles are to higher hierarchical level cost nodes up until the maximum level that EAS is to service. All cpu nodes must have the same highest level cost node. The phandle list must not be sparsely populated with handles to non-contiguous hierarchical levels.

In the example above, the Cortex-A57's sched-energy-costs node lists phandles labelled '&CPU_COST_0' and '&CLUSTER_COST_0'. Following the convention described in the previous paragraph, this is an ordered list with the first element being a phandle to a cost node that describes costing data for Cortex-A57 CPUs. The second element in the list is a phandle to a cost node that describes costing data for a cluster of Cortex-A57 CPUs.

Cost nodes are children of a special energy-costs parent node. Cost nodes contain two properties: a busy-cost-data property and an idle-cost-data property. These describe the previously introduced busy costs and idle costs. A busy-cost-data property is an array of 2-item tuples, each of type u32. The first item in the tuple is a capacity value and the second item in the tuple is the energy cost value associated with
that capacity. An idle-cost-data property is an array of 1-item tuples, each of type u32. The item in the tuple is the energy cost value associated with the idle state the item refers to.

The scheduler has been suitably extended to process the relevant portions of the FDT and extract the costing data. The data is then supplied to the EAS core code for further processing as described in the next section.

For a detailed specification of the bindings referred to here, see: Documentation/devicetree/bindings/scheduler/sched-energy-costs.txt.