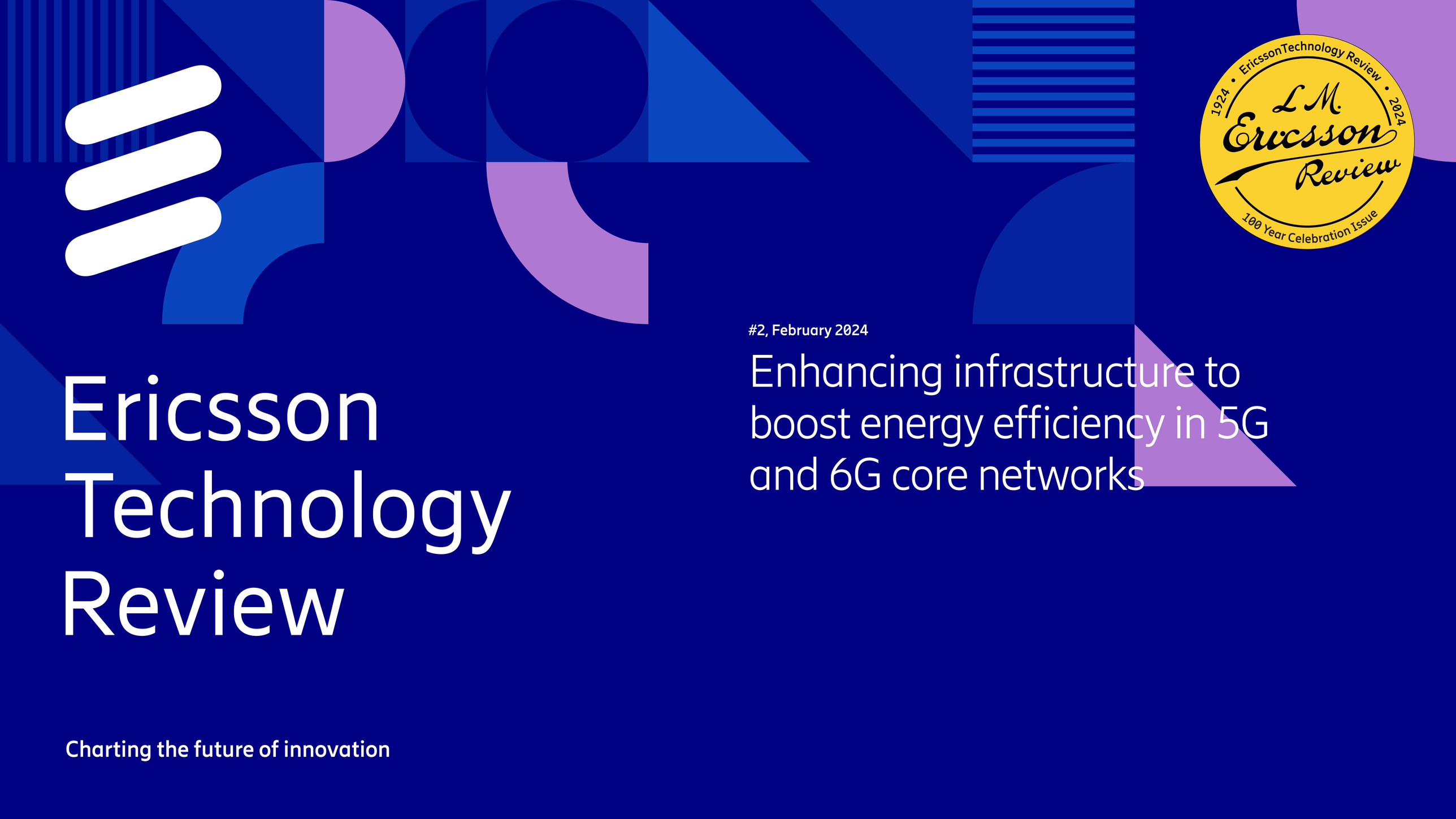

Ericsson Technology Review

Charting the future of innovation

#2, February 2024

Enhancing infrastructure to boost energy efficiency in 5G and 6G core networks

Ericsson Technology Review — Charting the future of innovation | #2, February 2024

Enhancing infrastructure to boost energy efficiency in 5G and 6G core networks

Authors:
Leif Johansson, Per Holmberg, Robert Skog

Communication service providers are keen to reduce energy consumption, enhance performance and reduce the need for large system configurations and power supplies in mobile core networks. To introduce the necessary application knowledge, we have developed a shim layer that can intercept and configure the relevant energy-saving mechanisms in the cloud environment.

ISSN 0014-0171 284 23-3405 | Uen

Ericsson, SE-164 83 Stockholm, Sweden
Phone: +46 10 719 0000

Energy-saving mechanisms can only work when they are applied correctly. In the cloud environment, application knowledge is essential to achieving energy savings without risking any application key performance indicators (KPIs) or modifying the Kubernetes software.

While Kubernetes-based, cloud-native deployments of mobile core networks deliver a variety of important benefits, they also limit power-management capabilities and hamper energy efficiency due to a lack of knowledge about the execution characteristics and requirements of individual applications. For example, an application running on one node may have lower KPI requirements in terms of latency and performance, while an application on another node is crucial in meeting the characteristic deadlines. If all computers and resources are deployed equally with the same power-management logic and without considering the applications' needs, the result is likely to be unnecessarily high energy consumption for low-performance tasks and reduced performance when energy is wasted on low-performance tasks.

Overview of energy-saving mechanisms in cloud infrastructure

Figure 1 shows a cloud-native deployment with a guaranteed container (in purple) that has exclusive central processing unit (CPU) cores (Core 0 and Core 1) next to burstable and best-effort containers (in light green) that share cores (Core 2 and Core 3). Each CPU core can be individually controlled through frequency scaling (P-state), halt states (C-state) and micro-sleeps. The energy consumption of the CPU fabric is controlled through uncore frequency, which affects the performance of all CPU cores as it is a shared resource.

To save energy without compromising application KPIs such as latency, the four available energy-saving mechanisms — core frequency scaling, uncore frequency scaling, micro-sleep and C-states — must be applied differently depending on the needs of each application.

Core frequency scaling

New CPU generations provide the ability to control the frequency on each core independently. Most mobile core applications are event-driven and require fast reactions to external network events to meet service latency requirements and service fulfillment. Measurements from mobile core networks show service requests arriving in bursts to the service handling node. A traffic-idle period can be suddenly followed by a burst of traffic events that require maximum computer performance to meet the service latency requirements.

The host operating system- (OS-)governed core frequency scaling has shown itself to be too slow to react to traffic bursts, which can lead to reduced system performance when enabled. Therefore, it is advisable to disable the host OS-governed core

Terms and abbreviations

API — Application Programming Interface | **CPU** — Central Processing Unit | **KPI** — Key Performance Indicator | **OS** — Operating System | **PCI** — Peripheral Component Interconnect | **QoS** - Quality of Service | **UPF** — User Plane Function

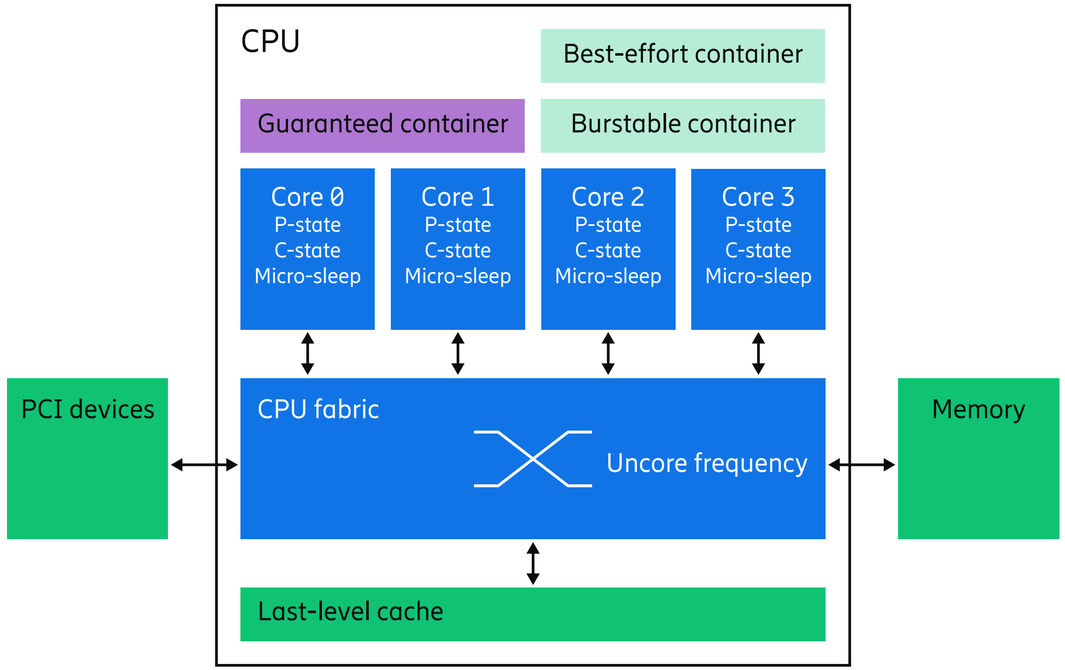

Figure 1: Cloud-native deployment example

frequency scaling when running mobile core applications. Mobile core cloud-native applications consist of several microservices, where each service is described in a chart, such as a Helm chart. Not all services require maximum undisturbed computer performance, as many are best-effort services that can make use of the host OS-governed core frequency scaling to save energy when the service is not in use. However, the host-OS-managed power core frequency treats all microservices equally and cannot distinguish between services that handle traffic and those that do not require high performance. As a consequence, best-effort services can cause false triggering of high core frequency and unnecessary energy consumption. To reduce energy consumption, it is important to configure the host OS not to scale up the performance for best-effort services, as this unnecessary high frequency results in higher energy consumption.

The results of the proof of concept presented later in this article indicate a notable increase in energy consumption – from 104W to 177W in our example – due to best-effort services. Therefore, it is crucial to optimize the implementation of host-OS-governed power management to ensure appropriate resource usage and prevent unnecessary energy consumption.

Uncore frequency scaling

The performance of the uncore, which includes the internal interconnect and parts of the processor shared by all cores, is crucial for memory and communication-bound applications. The interconnect performance is controlled by adjusting the uncore frequency. And by maintaining a low frequency, matching the actual needs, it saves CPU power.

However, if application characteristics and requirements are not taken into consideration, the default behavior is typically to scale up the uncore frequency just in case, even when a single core scales to high frequency. This often happens unnecessarily for CPU-bound programs that do not generate significant load on the CPU fabric, as well as background activities that are not critical and can use a lower and power-efficient uncore frequency. Even best-effort services can cause false triggering of high uncore frequency and unnecessarily high energy consumption.

Micro-sleep

Today, if an application needs to react quickly to traffic events, it continuously polls event queues in a busy loop, resulting in constant CPU execution at maximum performance and high energy consumption. To avoid such busy looping, micro-sleeps can be used to save energy between traffic events.

Micro-sleep is a technology that enables processor cores to cease execution until something new arrives in an event queue. The time required for waking up from a micro-sleep state is only a few hundred nanoseconds, allowing the application to save energy while still reacting quickly to new events. This can be particularly useful for packet processing nodes like the user plane function (UPF), which can utilize micro-sleep while waiting for new packets.

Unfortunately, the micro-sleep state cannot be detected by the host OS today. The host OS treats a CPU core that uses the micro-sleep state to save energy as 100 percent busy and never triggers core/uncore frequency scaling.

C-states

Processors provide lighter and deeper sleep modes (C-states) for cores and clusters of cores with shared cache, as well as for the full processor chip where a lighter sleep state, like C1, allows for a faster wake-up recovery but still has higher overhead than micro-sleep. Part of this comes from the software interface where sleep states are managed by the OS, giving context switch overhead and deeper sleep states that provide better power savings that also have further overhead in saving and restoring processor and cache states.

Proof of concept: Introducing application knowledge

In our proof of concept, we configured the Kubernetes CPU manager to provide guaranteed quality of service (QoS) class containers with an exclusive set of CPU cores, as shown in Figure 1. This approach ensures that containers or services with guaranteed performance can react quickly to network events, as the cores are not being used by anyone else. To reduce the complexity and achieve better hardware utilization, we chose a cloud-native deployment model using Kubernetes over bare metal infrastructure [1], which is the de facto standard.

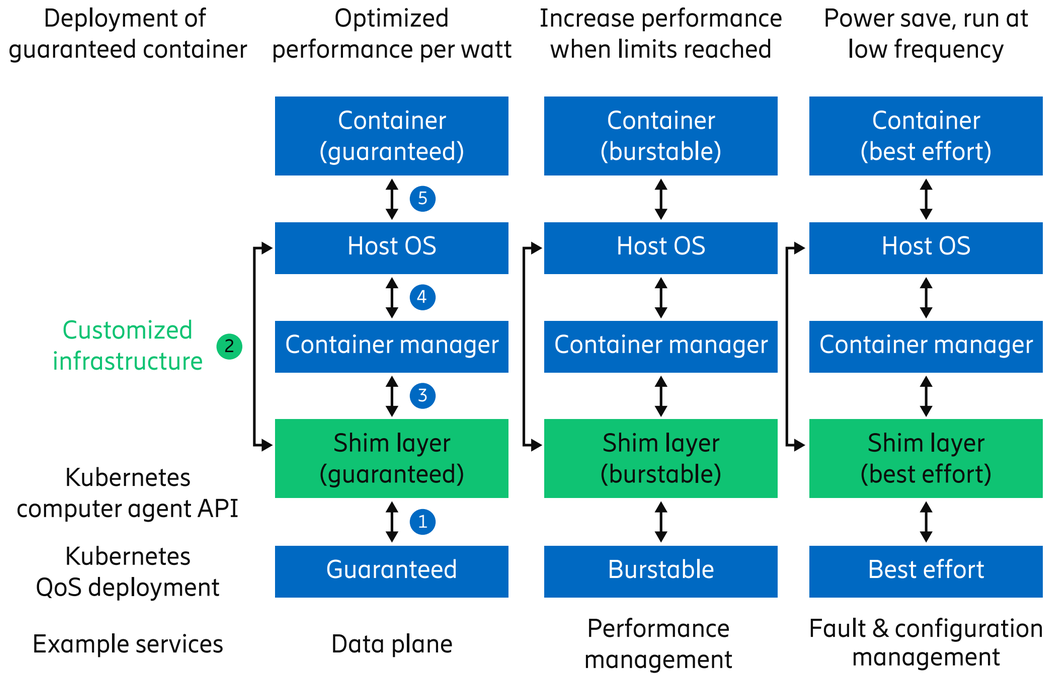

Figure 2: How the shim layer intercepts the container manager socket API

The Kubernetes application package manager we opted for is Helm, where the application deployment is described in Helm charts. The Helm charts contain various infrastructure requirements for each microservice, such as memory usage and CPU resources. Based on real-time characteristics, three different QoS classes can be specified in a Helm chart. Furthermore, a Helm chart can optionally include application hints to optimize power usage. In our research, we have added new application hints to configure the hardware infrastructure.

To achieve the goal of greater energy efficiency when deploying telecom-grade services on cloud-native infrastructure [1], we used a standard Kubernetes container deployer plus a new shim layer, as illustrated in **Figure 2**. The shim layer does not affect Kubernetes and is used to customize the infrastructure power-management functions before and after the container deployment, meeting KPI requirements for critical applications while optimizing energy efficiency.

Our method is based on Kubernetes QoS classes and additional information provided in application Helm charts to deliver application knowledge and application hints. Guaranteed QoS class is assumed in our experiment for services used in traffic execution that need deterministic performance to fulfill application KPIs. The burstable/best-effort QoS is used when the performance requirements and fast response times are more relaxed.

Our focus is therefore to reduce the energy consumption for services with more relaxed requirements (burstable and best-effort containers), and use the saved energy to boost the peak performance of guaranteed containers at traffic peaks, while simultaneously reducing energy consumption when the traffic is low. The shim layer then uses this information to configure the host OS power management, resulting in improved performance and energy efficiency of computer resources, while maintaining the application KPIs.

Setting up the shim layer

In the initial step, the shim layer is configured to customize the infrastructure based on the service type (guaranteed, burstable, best-effort and so on). The socket application programming interface (API), which Kubernetes uses to trigger the start of a new container, is redirected to the shim layer by changing the transport protocol port number used by the container manager to the port number used by the shim layer, marked as step 1 in Figure 2. The port number to start a container is part of the Kubernetes configuration files and can be changed by restarting the Kubernetes services.

Customizing the infrastructure

Before starting the new container, the shim layer starts to customize the infrastructure based on the Kubernetes QoS class and application hints provided in the application Helm charts, marked as step 2 in Figure 2. As an example, for a guaranteed container, we customized the infrastructure to set the frequency of the guaranteed non-shared CPU cores and uncore frequency as high as possible using the host OS API, to boost the performance for the guaranteed service.

We configured the shared CPU cores used for best-effort/burstable services to be managed by the host OS and execute at the lowest possible core frequency and at the lowest possible CPU halt state (C-state) when the cores are idle. The shared CPU cores are also configured to never enter opportunistic, not guaranteed frequencies (turbo frequencies) when busy to further save energy. This enables the system to save as much energy as possible for services that are not guaranteed and avoids unnecessary scale-up of the frequency for non-critical services.

For best-effort/burstable configured containers, we forwarded the request to the container manager without any change to the infrastructure, which was already in the lowest power-manager state.

Kubernetes scales burstable containers based on resource utilization during runtime. The resource scaling request is intercepted by the shim layer, and it triggers new infrastructure settings based on the resource demands. For example, when resource demands increase, it will scale up the CPU core frequency.

Application hints

The optionally provided application hints help the shim layer prioritize critical application resources. In our work, we have introduced two application hints: prioritized core frequency and prioritized uncore frequency.

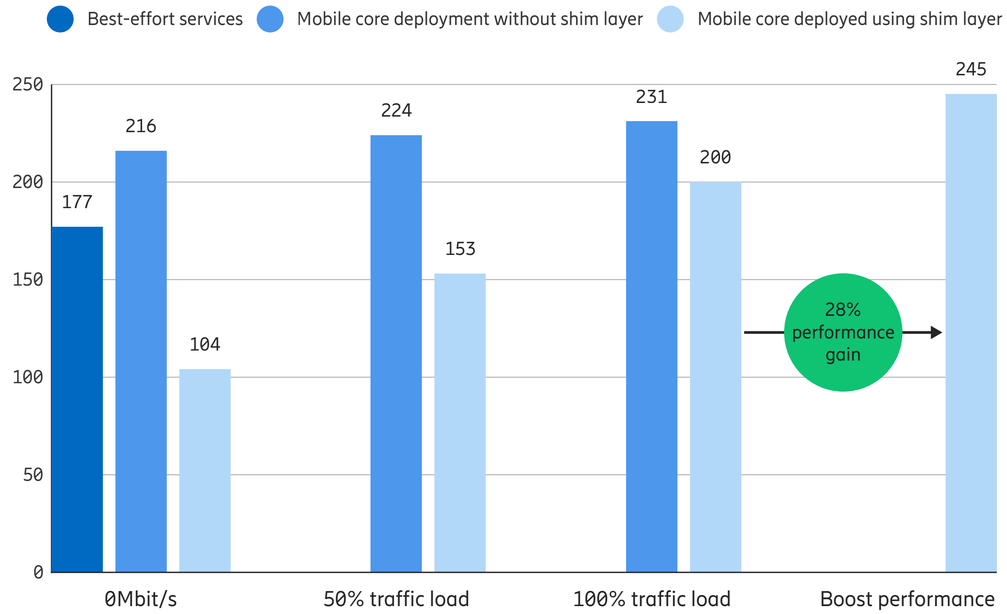

Figure 3: Mobile core CPU power consumption

The core frequency hint is used when the application performance scales with the CPU core clock frequency and is less dependent on the uncore frequency. To achieve maximum performance or improve performance per watt, the shim layer reduces the uncore frequency and scales up the CPU core frequency when the core frequency hint is applied.

The uncore frequency hint is used when the application scales with uncore frequency – for example, applications streaming data from the memory system. For uncore frequency application hints, the shim layer reduces the core frequency and scales up the uncore frequency.

Saving energy at low traffic

To save energy and react quickly to traffic events, micro-sleep is used as soon as the CPU core is idle when running user-plane containers with guaranteed class of service [2]. For services where network events are provided by the host OS and configured with guaranteed QoS class, we configure the host OS to use the fast CPU halt state (C-State 1E) as soon as the guaranteed non-shared core is idle. To further save energy, the core frequency is scaled down to its minimum when the core is idle and scaled up to its maximum core frequency when active (rush-to-idle policy). This enables the application to trigger maximum energy savings when idle and deterministic high performance when active. The method can be applied without affecting any application KPIs.

Uncore frequency scaling

To avoid high CPU fabric energy consumption, only services with guaranteed QoS class trigger the uncore frequency. In our tests, the guaranteed non-shared CPU core with the highest utilization determines the uncore frequency. Note that the CPU fabric is a shared resource, and the uncore frequency impacts interconnect performance on all CPU cores. The method ensures that non-critical services do not trigger unnecessarily high CPU fabric energy consumption, while critical CPU cores receive the needed performance to fulfill application KPIs.

Forwarding the container request

After the infrastructure customization, the container request is forwarded to the container manager, indicated as step 3 in Figure 2. The guaranteed service is started on the cores assigned to the container (shown in steps 4 and 5). The system is now performance and energy optimized without impacting application KPIs.

Proof-of-concept experimental validation

We validated the enhanced cloud infrastructure by deploying 5G mobile applications of the user plane function (UPF) type. We then generated a realistic workload and applied it to the deployed UPF applications.

All measurements are based on a dual-socket server equipped with Intel Sapphire Rapid CPUs with a power limit of 300W. Only one of the CPUs is used for traffic handling. All figures are based on the CPU socket used for traffic. In our experiments, we measured characteristics when deploying the UPF on our enhanced cloud infrastructure and compared the results when deployed in a standard Kubernetes cluster. The DPDK (Data Plane Development Kit) based UPF uses a micro-sleep function specially designed to save energy when used in combination with the shim layer [2]. Note that the OS kernel is not involved at all when processing packets.

The shim layer reduces the uncore frequency and scales up the CPU core frequency.

As a traffic model, we used a traffic mix from our internal stability test. We generated the traffic using an external traffic generator that simulates the surrounding network elements.

Proof-of-concept results

Figure 3 shows the processor power consumption, with the medium blue bars demonstrating the power usage when running the unmodified mobile core without the shim layer at idle load (0Mbit/s), at 50 percent of the maximum traffic load and at 100 percent of the maximum traffic load. The first three light blue bars represent the energy consumption when using the energy-optimized mobile core with the shim layer. The dark blue bar to the far left of the figure illustrates the energy usage for best-effort services when deployed on a standard Kubernetes cluster. The light blue bar to the far right highlights the results when using the "boost performance" feature, which utilizes the saved energy to boost CPU performance.

The dark blue bar in Figure 3 indicates that best-effort services, which are not used in traffic and are not real-time critical, consume 177W of energy when deployed using the standard Kubernetes cluster, which is significantly higher than the 104W required when using the energy-optimized mobile core with the shim layer. The reason for this is that the standard Kubernetes deployment cannot differentiate between performance-critical microservices and best-effort microservices, resulting in unnecessarily high energy consumption.

Our results further indicate that processor energy consumption can be reduced by more than 50 percent at idle load when using the shim layer. The energy consumption increases as the traffic load grows but never exceeds the CPU power limit of 300W at maximum traffic load. The energy saved is used to further boost mobile core performance – that is, boost the uncore and CPU core frequency – which provides 28 percent additional performance gain, as illustrated by the light blue bar to the far right of Figure 3. The performance boost can be safely applied within the 300W CPU power budget using the shim layer.

Key benefits

It is clear from our results that the use of separate policies for different classes of workloads enables substantial energy savings. Data plane services are set up to use a rush-to-idle policy and opportunistically enter micro-sleep state in any possible gap between packets. Best-effort services are instead set up to use an energy-efficient policy that does not drive high power by using inefficient turbo-boost frequencies. Together, these policies create a power headroom that can be used to allow higher core frequency for cores that are used for traffic handling. This enables a higher peak capacity and allows a higher system performance within the same configuration and power envelope.

Processor energy consumption can be reduced by more than 50 percent at idle load.

The energy-saving mechanisms enabled by enhanced cloud infrastructure (especially micro-sleeps) prevent the processors from reaching high temperatures that negatively impact server lifetime. They also prevent performance degradation caused by power limitation at workloads as high as 80 percent processor utilization. Higher peak frequency on the traffic-handling services enables energy-efficient boost peak capacity.

Our measurements also demonstrate that improved power efficiency makes it possible to reduce the amount of hardware by 30 percent while maintaining application KPIs. A smaller system configuration with fewer processors is more energy efficient in and of itself. While the processor power-management mechanisms are efficient in saving energy in the processor, the smaller configuration saves the full energy of the servers.

Conclusion

Our research indicates that it is possible to both save energy and boost performance by implementing an enhanced cloud infrastructure. Access to application knowledge – and application hints – enables the introduction of energy-saving functions in the cloud environment without putting the key performance indicators of applications at risk or requiring modifications to the Kubernetes software. In our proof of concept, we introduced application knowledge in the cloud environment in the form of a shim layer that intercepts and configures the appropriate energy-saving mechanisms. With this approach, we achieved central processing unit energy savings of more than 50 percent at low traffic, a boost in peak performance and a 30 percent reduction in hardware.

The authors

Leif Johansson is an expert in the characteristics of open systems. He joined Ericsson in 1996 after receiving an M.Sc. in physics from Uppsala University, Sweden. Johansson has more than 20 years of experience evaluating and applying new technology in Ericsson's product portfolio.

Per Holmberg worked at Ericsson from 1985 to 2023, most recently as a computer architect designing processing solutions for communication equipment. During his career at Ericsson, Holmberg held specialist and expert positions in computer architecture and processing systems, as well as serving as lead architect for several processor and computer designs. He is responsible for more than 50 inventions. Holmberg holds an M.Sc. in electrical engineering from KTH Royal Institute of Technology in Stockholm, Sweden.

Robert Skog worked at Ericsson from 1989 to 2023, most recently as a senior expert in the field of service architecture. During his career at Ericsson, his work focused on end-to-end solutions and traffic optimization for everything from the first wireless application protocol solutions to today's advanced user plane solutions. In 2005, Skog won Ericsson's prestigious Inventor of the Year award. He holds an M.Sc. in electrical engineering from KTH Royal Institute of Technology.

References

1. Ericsson blog, Why Kubernetes over bare metal infrastructure is optimal for cloud native applications, May 3, 2022, Bäckström, H; Bohra, R ↗
2. Ericsson Technology Review, Energy-efficient packet processing in 5G mobile systems, June 21, 2022, Johansson, L; Holmberg, P; Skog, R ↗

Further reading

- Ericsson, Network intelligence and services ↗
- Ericsson blog, Achieving sustainability with energy efficiency in 5G networks ↗
- Ericsson blog, Finding the right balance between energy savings and network performance ↗
- Ericsson blog, Sustainable networks: Saving energy while maintaining customer experience ↗
- Ericsson Technology Review, Ensuring energy-efficient networks with artificial intelligence ↗