Technology Guide

Power Manager – Kubernetes Operator

Authors

Philip Brownlow

Patricia Cahill

Lukasz Danilczuk

1 Introduction

In a container orchestration engine such as Kubernetes (K8s), the allocation of CPU resources from a pool of platforms is based solely on availability. There is no consideration for individual capabilities such as Intel® Speed Select Technology (Intel® SST), Intel® Advanced Vector Extensions (Intel® AVX), Intel® QuickAssist Technology (Intel® QAT), or Intel® Advanced Encryption Standard New Instructions (Intel® AES-NI).

This document describes the Kubernetes Power Manager, a Kubernetes operator developed using the Operator SDK, designed to expose and use Intel-specific power management technologies in a Kubernetes environment.

Intel SST is a powerful collection of features that offers more granular control over CPU performance and power consumption on a per-core basis. However, as a workload orchestrator, Kubernetes is intentionally designed to provide a layer of abstraction between the workload and such hardware capabilities. This presents a challenge to Kubernetes users running performance-critical workloads with specific requirements that are dependent on hardware capabilities.

The Kubernetes Power Manager bridges the gap between the container orchestration layer and hardware features enablement, specifically Intel SST, by allowing the user to tune the frequencies and set the priority level of the cores chosen by the Kubernetes Native CPU Manager.

This document is intended for customers and engineers looking to optimize performance and power efficiency on the latest Intel® Xeon® Scalable processors in the K8s cluster.

This document is part of the Network Transformation Experience Kits.

NOTE: The general information contained within this document applies to both the 3rd Gen Intel® Xeon® Scalable processor and the Intel® Xeon® D processor. However, please note that the performance, configurations, and feature set in this document apply specifically to the 3rd Gen Intel® Xeon® Scalable processor and may vary for the Intel® Xeon® D processor.

Table of Contents

Figures

Tables

Document Revision History

Revision	Date	Description
001	October 2021	Initial release.
002	February 2022	Added a note regarding Intel® Xeon® D processor in the Introduction section. Updated information regarding Base PowerProfile in Section 3.4.
003	October 2022	An updated version of the Kubernetes Power Manager v2 was released in August 2022. The latest release introduces the newly developed Intel® Power Optimization Library as a replacement for the previously used AppQoS suite. Also, the new release has the addition of C-state, P-state governor, and Time of Day functionality.

1.1 Terminology

Table 1. Terminology

Abbreviation	Description
ACPI	Advanced Configuration and Power Interface
API	Application Programming Interface
CR	Custom Resource
CRD	Custom Resource Definition
EPP	Energy Performance Preference is the value that associates a core with a priority level when using Intel® Speed Select Technology – Core Power (Intel® SST-CP).
Exclusive CPU	An entire physical core dedicated exclusively to the requesting container, which means no other container has access to the core. Assigned by the exclusive pool within CPU Manager for Kubernetes.
Exclusive Pool	A group of isolated, exclusive CPUs where a container is exclusively allocated the requested number of CPUs, meaning only that container can run on that CPU.
Intel® SST-BF	Intel® Speed Select Technology – Base Frequency (Intel® SST-BF)
Intel® SST-CP	Intel® Speed Select Technology – Core Power (Intel® SST-CP)
Intel® SST-TF	Intel® Speed Select Technology – Turbo Frequencies (Intel® SST-TF)
K8s	Kubernetes
NFV	Network Function Virtualization
OVS	Open vSwitch
PCU	Power Control Unit
Pool	CPU Manager for Kubernetes uses a Kubernetes config-map to represent the cores available on the system. The items in this config-map are defined as pool. A pool, in this context, is a named group of CPU lists.
RBAC	Role-Based Access Control
SDK	Software Development Kit
Shared Pool	A group of isolated, shared CPUs where a requesting container can run on any CPU in this pool with no guaranteed exclusivity.

1.2 Reference Documentation

Table 2. Reference Documents

Reference	Source
Kubernetes Power Manager Repository	https://github.com/intel/kubernetes-power-manager
Red Hat: What is a Kubernetes Operator?	https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-operator
Extending Kubernetes	https://kubernetes.io/docs/concepts/extend-kubernetes/
Operator Pattern	https://kubernetes.io/docs/concepts/extend-kubernetes/operator
Kubebuilder	https://book.kubebuilder.io/
Operator Framework	https://operatorframework.io/
OperatorHub.io	https://operatorhub.io/
CommsPowerManagement	https://github.com/intel/CommsPowerManagement
Intel® Speed Select Technology – Base Frequency (Intel® SST-BF) with Kubernetes Application Note	https://builders.intel.com/docs/networkbuilders/intel-speed-select-technology-base-frequency-with-kubernetes-application-note.pdf
Kubernetes Operators – Automated Lifecycle Management Technology Guide	https://networkbuilders.intel.com/solutionslibrary/kubernetes-operators-automated-lifecycle-management-technology-guide
Intel® Power Optimization Library	https://github.com/intel/power-optimization-library
Power Management – Technology Overview Technology Guide	https://builders.intel.com/docs/networkbuilders/power-management-technology-overview-technology-guide.pdf

2 Overview

2.1 Challenges Addressed

Today's diverse range of workloads with varying usage and performance demands has outgrown standard CPU capabilities. This presents a challenge to Kubernetes's users running performance-critical workloads with specific requirements dependent on hardware capabilities. The Kubernetes Power Manager, as a workload orchestrator, is intentionally designed to provide a layer of abstraction between the workload and such hardware capabilities.

2.2 Use Cases

Among the use cases that the Kubernetes Power Manager addresses the following are just a few:

- NFV infrastructure containers such as Open vSwitch (OVS)
  Ensure priority CPUs (i.e., CPUs that can operate at a guaranteed higher frequency) are used for packet processing activity.
- Frequency bound workloads such as software-based crypto
  Software-based crypto applications require high frequency cores to ensure predictable performance. Dynamic CPU frequency tuning can help ensure that these workloads meet performance SLAs.
- Data center power consumption optimization
  Scale up and down CPU frequencies and power usage based on workload demand, dynamically accommodating usage spikes and lulls due to varying demand over time. The user may want to preschedule nodes to move to a performance PowerProfile during peak times to minimize spin up. At times during off-peak, the user may also want to move to a power-saving PowerProfile.
- Unpredictable machine use
  May use machine learning through monitoring to determine PowerProfiles that predict a peak need for compute, to spin up ahead of time.
- Power optimization over performance
  A cloud-based system may be interested in fast response time, but not in maximal response time, so may choose to spin up cores on demand, and only those cores, but want to remain in power-saving mode the rest of the time.

2.3 Technology Description

The Kubernetes Power Manager follows the Kubernetes operator pattern that is now commonplace for system configuration and application deployment in the K8s ecosystem. For a general overview of the operator pattern in K8s, refer to Kubernetes Operators – Automated Lifecycle Management Technology Guide.

2.3.1 Intel Power Optimization Library

The Intel Power Optimization Library is an open-source library that takes the desired configuration of the user to tune the frequencies and set the priority level of the cores.

The Intel Power Optimization Library takes the desired configuration for the cores associated with Exclusive Pods and tunes them based on the requested Power Profile. The Intel Power Optimization Library also facilitates the use of the Intel SST suite (Intel SST-BF, Intel SST-CP, and Intel SST-TF). The Intel Power Optimization Library also facilitates C-state enablement, allowing the user to have more granular control over "sleep state" in the system.

2.3.2 Node Agent

The node agent is a containerized application deployed by the Kubernetes Power Manager in a DaemonSet. The primary function of the node agent is to communicate with the node's Kubelet PodResources endpoint to discover the exact CPUs that are allocated per container. The node agent watches for pods that are created in the cluster, examines them to determine which PowerProfile they have requested, and then sets off the chain of events that tunes the frequencies of the cores designated to the pod.

3 Kubernetes Power Manager Components

The Kubernetes Power Manager has two main components, the Overarching Operator and the Power Node Agent.

The Overarching Operator is responsible for the configuration and deployment of the Power Node Agent, while the Power Node Agent is responsible for the tuning of the cores as requested by the user by communicating with the Intel Power Optimization Library.

3.1 Kubernetes Power Manager Controller

It is the function of the Kubernetes Power Manager controller to compare the desired state of the cluster and its actual state. If the desired state and the actual state do not match, it is the job of the controller to act and amend the problem.

3.1.1 PowerConfig Controller

The PowerConfig Controller waits for the PowerConfig to be created by the user, in which the desired PowerProfiles are specified. The PowerConfig holds different values: What nodes the user wants to place the node agent on, and what PowerProfiles are required.

- `powerNodeSelector`: A key/value map used to define a list of node labels that a node must satisfy for the operator's node agent to be deployed.
- `powerProfiles`: The list of PowerProfiles that the user wants available on the nodes.

After the PowerConfig Controller sees that the PowerConfig is created, it reads the values and then deploys the Power Node Agent onto each of the nodes that are specified using a DaemonSet. Then it creates the PowerProfiles and Extended Resources. Extended Resources are resources created in the cluster that can be requested in the PodSpec. The Kubelet keeps track of these requests. Extended Resources are important to use as they can specify how many cores on the system can be run at a higher frequency before hitting the heat threshold.

The PowerConfig status represents the nodes that match the `powerNodeSelector` and, as such, have the Power Node Agent deployed.

The following is a PowerConfig example. Only one PowerConfig object is permitted per cluster. The PowerConfig Controller does not allow a second configuration to be applied.

```
apiVersion: power.intel.com/v1
kind: PowerConfig
metadata:
  name: power-config
  namespace: intel-power
spec:
  powerNodeSelector:
    # Add labels here for the Nodes you want the PowerNodeAgent to be applied to
    feature.node.kubernetes.io/power-node: "true"
  powerProfiles:
  # Add wanted PowerProfiles here; valid entries are as follows:
  # performance
  # balance-performance
  # balance-power
  - "performance"
```

3.2 Power Node Agent

The Power Node Agent is deployed to each of the nodes requested by the user in the PowerConfig. Each pod contains a container to host the Power Node Agent controllers.

The Power Node Agent holds the following components: PowerProfile Controller, PowerWorkload Controller, PowerNode Controller, and Pod Controller

3.2.1 PowerProfile Controller

The PowerProfile custom resource (CR) holds values for specific Intel SST settings that are then applied to CPUs at host level by the operator as requested. PowerProfiles are advertised as Extended Resources and can be requested via the PodSpec. A PowerProfile is created in the cluster, which the PowerProfile controller uses to create an instance of in the Intel Power Optimization Library in the form of a Profile. Three default PowerProfiles can be created via the PowerConfig: "performance," "balance-performance," and "balance-power." These PowerProfiles are not given a Max or Min value. Instead, those values are calculated by the PowerProfile controller upon creation, which means the values vary across nodes in a cluster. A user may also create a PowerProfile, but the Max and Min values must be provided, and they also must not be outside of the CPU's capacity. The following show PowerProfile examples. A PowerProfile is requested via the PodSpec.

PowerProfile

```
apiVersion: power.intel.com/v1
kind: PowerProfile
metadata:
    name: example-power-profile
    Namespace: intel-power
spec:
  name: "example-power-profile"
  maxFrequency: 2800
  minFrequency: 2600
  epp: "performance"
```

Shared PowerProfile (created by user)

```
apiVersion: power.intel.com/v1
kind: PowerProfile
metadata:
    name: shared
   Namespace: intel-power
spec:
  name: "shared"
  maxFrequency: 1100
  minFrequency: 1100
  epp: "power"
```

The following figure shows an example of a PowerProfile workflow, from creation at the cluster level to the PowerProfile controller creating a profile in the Intel Power Optimization Library.

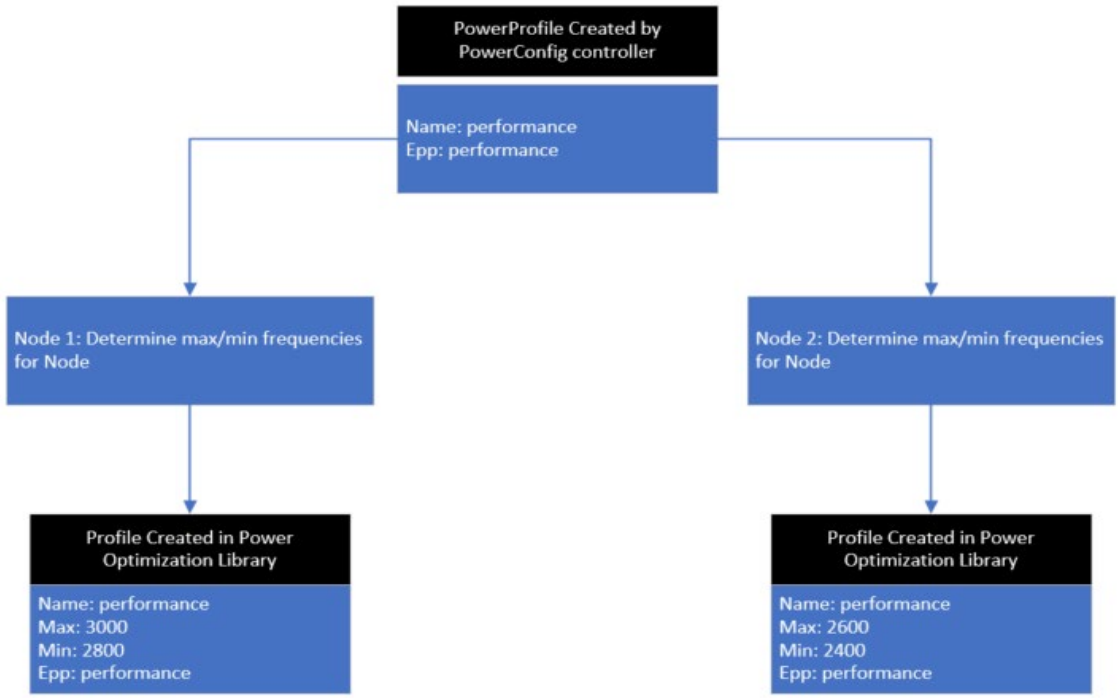

Figure 1. Profile Created with Intel Power Optimization Library

3.2.2 PowerWorkload Controller

The PowerWorkload is the object used to define the lists of CPUs configured with a particular PowerProfile. A PowerWorkload is created for each PowerProfile on each Node with the Power Node Agent deployed.

PowerWorkload objects are created automatically by the PowerProfile Controller. This action is undertaken by the Power Node Agent when a PowerProfile is created.

A PowerWorkload is represented in the Intel Power Optimization Library by a Pool. The Pools hold the values of the PowerProfile used, their frequencies, and the CPUs that need to be configured. The creation of the Pool – and any additions to the Pool – then carries out the changes.

PowerWorkload objects also can be created directly by the user via the PowerWorkload spec. The user creates a PowerWorkload in the instance of a shared PowerWorkload.

3.2.2.1 PowerWorkload Created Directly by User

PowerWorkload objects can be created directly by the user via the PowerWorkload spec. This is only recommended for configuring the Native CPU Manager's shared pool.

It is not recommended to directly configure PowerWorkloads with specific nodes and CPU IDs. Instead, directly configuring PowerWorkloads should be done by using the `powerNodeSelector` and `reservedCPUs` options, as shown in the following example.

```
apiVersion: power.intel.com/v1
kind: PowerWorkload
metadata:
  # Replace <NODE_NAME> with the Node you intend this PowerWorkload to be associated with
  name: shared-<NODE_NAME>-workload
  namespace: intel-power
spec:
  # Replace <NODE_NAME> with the Node you intend this PowerWorkload to be associated with
  name: "shared-<NODE_NAME>-workload"
  # The 'allCores: true' option signifies to the PowerWorkload Controller that this is a
  # Shared PowerWorkload
  allCores: true
  reservedCPUs:
  # IMPORTANT: The CPUs in reservedCPUs should match the value of the reserved system CPUs in
your Kubelet config file
  - 0
  powerNodeSelector:
    # The label must be as below, as this workload will be specific to the Node
    kubernetes.io/hostname: <NODE_NAME>
  # Replace this value with the intended shared PowerProfile
  powerProfile: "shared"
```

This PowerWorkload assigns the shared PowerProfile to all CPUs in the Native CPU Manager's shared pool (minus those specified in the `reservedCPUs` field) on nodes that match the `powerNodeSelector` labels.

The `reservedCPUs` option is used to represent the list of CPUs that have been reserved by the Kubelet. With this option the user only needs to configure CPUs that are exclusively allocatable (i.e., shared pool – reserved CPUs) to containers with a given PowerProfile. This allows the reserved CPUs to continue with default settings, exempt from any PowerProfile.

If `powerNodeSelector` is specified and a node list is also specified, `powerNodeSelector` takes precedence and the specified nodes list is redundant.

Note: A PowerWorkload can satisfy only one node in the user's cluster, so when specifying labels for the `powerNodeSelector` you should be as specific as possible. For example, use the `kubernetes.io/hostname=<NODE_NAME>` label, as this can be matched only by a single node. If a shared PowerProfile is to be used by multiple nodes in the cluster, a separate PowerWorkload should be created for each node using the same PowerProfile.

The following example workload assigns the performance PowerProfile to cores 5, 6 on node1 and cores 10, 12 on node2.

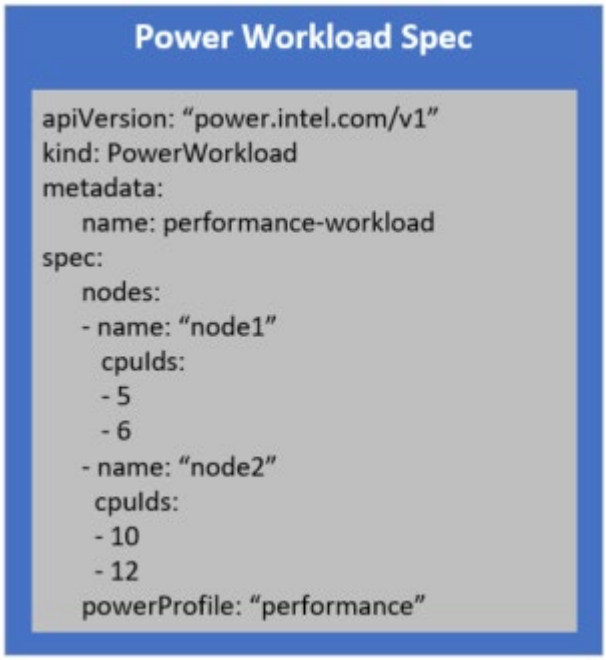

Figure 2. PowerWorkload Spec Example

3.2.3 PowerNode Controller

A PowerNode is created for each node in the cluster that matches the `powerNodeSelector` labels in the PowerConfig object. The purpose of this object is to allow the user to view which PowerProfiles are being used, what cores are being used, and the containers to which they are assigned. The two shared pools can be the default pool or the shared pool. If there is no shared PowerWorkload associated with the node, then the default pool holds all the cores in the cluster's 'shared pool', none of which will have their frequencies tuned to a lower value. If a shared PowerWorkload is associated with the node, then the cores in the shared pool are those in the cluster's 'shared pool' – excluding cores reserved for Kubernetes processes (`reservedCPUs`).

Each PowerNode object is named according to its corresponding node.

Use the following command to list all PowerNodes on the cluster.

```
kubectl get powernodes -A
```

Use the following command to display a specific PowerNode, such as the example above.

```
kubectl describe powernode worker-node-1 -n intel-power
```

The following example displays the PowerNode for `worker-node-1`.

```
name:         worker-node-1
namespace:    intel-power
apiVersion:  power.intel.com/v1
kind:         PowerNode
spec:
  nodeName:      worker-node-1


  Power Profiles:
    shared: 1000000 || 1000000 || power
    performance: 3000000 || 2800000 || performance
    balance-performance: 2450000 || 2250000 || balance_performance
    balance-power: 1900000 || 1700000 || balance_power
  Shared Pool:       shared || 1000000 || 1000000 || 1-86
```

Unaffected Cores: 0 The following example shows a PowerWorkload.

```
Name:         performance-<NODE_NAME>
Namespace:    intel-power
API Version:  power.intel.com/v1
Kind:         PowerWorkload
    Manager:         nodeagent
Spec:
  Name:           performance-<NODE_NAME>
```

```
Power Profile:  performance
Workload Nodes:
  Containers:
    Exclusive Cpus:
      1
      2
      3
      4
      49
      50
      51
      52
    Id:              containerd://XXX.XXXXX.XXXXX
    Name:            example-performance-container
    Pod:             example-performance-pod
    Power Profile:   performance
    Exclusive Cpus:
      1
      2
      3
      4
      49
      50
      51
      52
    Id:              containerd://XXX.XXXX.XXXXX
    Name:            example-performance-container
    Pod:             example-performance-pod
    Power Profile:   performance
  Cpu Ids:
    1
    2
    3
    4
    49
    50
    51
    52
  Name:  <NODE_NAME>
```

3.3 Pod Controller

The Pod Controller watches for pods. When a pod comes along, the Pod Controller checks if the pod is in the Guaranteed Quality of Service class (using exclusive cores), in which case it takes a core out of the shared pool. It is the only option in Kubernetes that can do this operation. Then it examines the pod's containers to determine which PowerProfile has been requested and creates or updates the appropriate PowerWorkload.

Note: The pod's requests and the limits must have a matching number of cores and PowerProfiles. Only one PowerProfile can be used in a single pod. If two are selected, the pod is created but no frequency tuning occurs.

3.4 Power PodSpec

Figure 3 shows that four CPUs are being requested with a performance PowerProfile attached.

Power Pod Spec

```
apiVersion: v1
kind: Pod
metadata:
    name: example-pod
spec:
    containers:
    - name: example-container
      image: ubuntu
      command: ["/bin/bash"]
      args: ["-c", "sleep 15000"]
      resources:
          requests:
              memory: "200Mi"
              cpu: "4"
              power.intel.com/performance: "4"
          limits:
              memory: "200Mi"
              cpu: "4"
              power.intel.com/performance: "4"
```

Figure 3. PodSpec Example Workflow

Figure 4 illustrates the execution workflow.

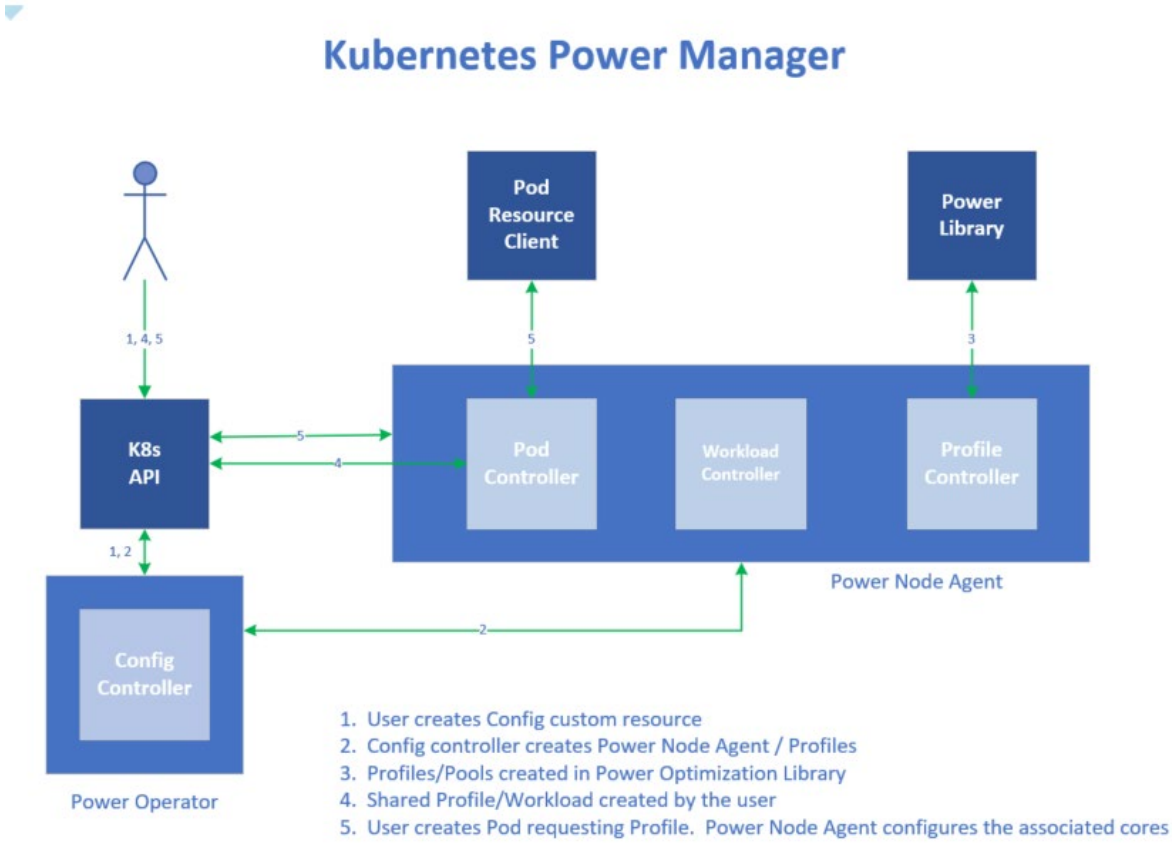

Figure 4. Execution Flow Diagram

The following describes each workflow step in Figure 4.

- The PowerConfig Controller sees that the user created PowerConfig CRD and determines the desired nodes for the Kubernetes Power Manager.
- The PowerConfig Controller updates the node status for each of these nodes in the Kubernetes API.
- The PowerConfig Controller publishes the PodSpec to the Kubernetes API for the Power Node Agent.
- The Kubelet creates the pods for the agent on the desired nodes.
- The PowerConfig Controller creates the PowerProfile CRDs that were requested in the PowerConfig Controller.
- The PowerProfile Controller sees the created PowerProfile CRDs and creates the corresponding Profile in the Intel Power Optimization Library, along with a PowerWorkload for that specific Node.
- The user creates a shared PowerProfile along with a shared PowerWorkload for the nodes in the cluster. The PowerWorkload Controller recognizes this workload as shared and tunes the cores in the shared pool of each appropriate node.
- The user deploys a pod that requests a PowerProfile.
- The Pod Controller sees this pod, determines the PowerProfile it wants to use, then updates the corresponding PowerWorkload CRD in the Kubernetes API.
- The PowerWorkload Controller sees the updated PowerWorkload CRD and updates the corresponding pool object in the Intel Power Optimization Library on the appropriate node.
- The Intel Power Optimization Library tunes the frequency of the required cores on its nodes.

4 Functionality

In-depth information about the following features is in the Experience Kits on Intel Network Builder: Network Transformation Experience Kits, Container Experience Kits, and 3rd Gen Intel® Xeon® Scalable processors Experience Kits.

4.1 Intel Speed Select Technology – Base Frequency (Intel SST-BF)

Intel SST-BF can control the base frequency of certain cores. The base frequency is designed for a guaranteed level of performance on the CPU (a CPU never goes below its base frequency). Priority cores can be set to a higher base frequency than a majority of the other cores on the system to which the user can apply their critical workload for a more guaranteed performance.[1]

4.2 Intel Speed Select Technology – Core Power (Intel SST-CP)

Intel SST-CP can group cores into levels of priority. When there is power to spare on the system, it can be distributed among the cores based on their priority level. There are four levels of priority available:

- Performance
- Balance Performance
- Balance Power
- Power

The priority level for a core is defined using its energy performance preference (EPP) value, which is one of the options in the PowerProfile CRD. If not all of the power is used on the CPU, the CPU can put the higher priority cores up to turbo frequency, which allows the CPUs to run faster.

4.3 Frequency Tuning

Frequency tuning allows the individual cores on the system to be sped up or slowed down by changing their frequency. This tuning is done via the Intel Power Optimization Library. The min and max values for a core are defined in the PowerProfile CRD, and the tuning is done after the core has been assigned by the Native CPU Manager. The frequency of a core is changed by writing the new frequency value to the `/sys/devices/system/cpu/cpuN/cpufreq/scaling_max|min_freq` file for the given core.

4.4 C-states

C-states are power states that a CPU can use to reduce power consumption on a per-core level, or on a CPU package level, by powering down portions of the core, package, or both. Disabling portions of the core allows for large power savings but prevents the core from executing instructions.

Table 3. C-state Ranges and Core C-states

C-state	Description
C0	Operating State
C1	Halt
C1E	Enhanced Halt
C2	Stop Grant
C2E	Extended Stop Grant
C3	Deep Sleep
C4	Deeper Sleep
C4E/C5	Enhanced Deeper Sleep
C6	Deep Power Down

- C0 is the active state where instructions are executed. No instructions are executed in other Core C-states.
- C-states range from C0 to Cn. C0 indicates an active state. All other C-states (C1-Cn) represent idle sleep states where the processor clock is inactive (cannot execute instructions) and different parts of the processor are powered down. As the C-states get deeper, the exit latency duration becomes longer (the time to transition to C0) and the power savings becomes greater.
- Lower C-states (C6 being the lowest) are power optimized, resulting in greater power savings and higher exit latency.
- If a core reaches C6, the L1 and L2 Caches are flushed to L3 Cache and data may need to be reloaded into cache after the core exits the power-optimized state.

[1] For workloads and configurations visit Performance Index. Results may vary.

4.4.1 C-state Management

Hardware: In certain configurations, the Power Control Unit (PCU) in the CPU is responsible for autonomously coordinating core and package C-states while BIOS configuration allows you to limit the C-states available to the platform, ensuring it never goes below your required C-state. Alternatively, `intel_idle` and the Linux scheduler govern C-states.

Operating System: Core C-states are controlled by the OS as defined by the ACPI Specification. The OS can tell the threads in a core to go to a particular C-state using the MWAIT and MONITOR instructions. In supported CPUs, application software can use the waitpkg instructions to request power-optimized states C0.1 and C0.2.

4.5 P-state Governor Functionality

The P-state governor feature allows the user to check if the P-state driver is enabled on the system. If the P-state driver is enabled while using the Kubernetes Power Manger, users may select a P-state governor per core, which are described as "performance" and "powersave" governors in the Power Profiles.

- Performance governor - The CPUfreq governor "performance" sets the CPU statically to the highest frequency within the borders of `scaling_min_freq` and `scaling_max_freq`.
- Powersave governor - The CPUfreq governor "powersave" sets the CPU statically to the lowest frequency within the borders of `scaling_min_freq` and `scaling_max_freq`.

The flow diagram in Figure 5 indicates the steps that a user invokes to use the P-state driver functionality on a system.

The user first checks that the P-state driver is available on the system. If available, the user then has the option to select the desired governor, either "Powersave" or "Performance".

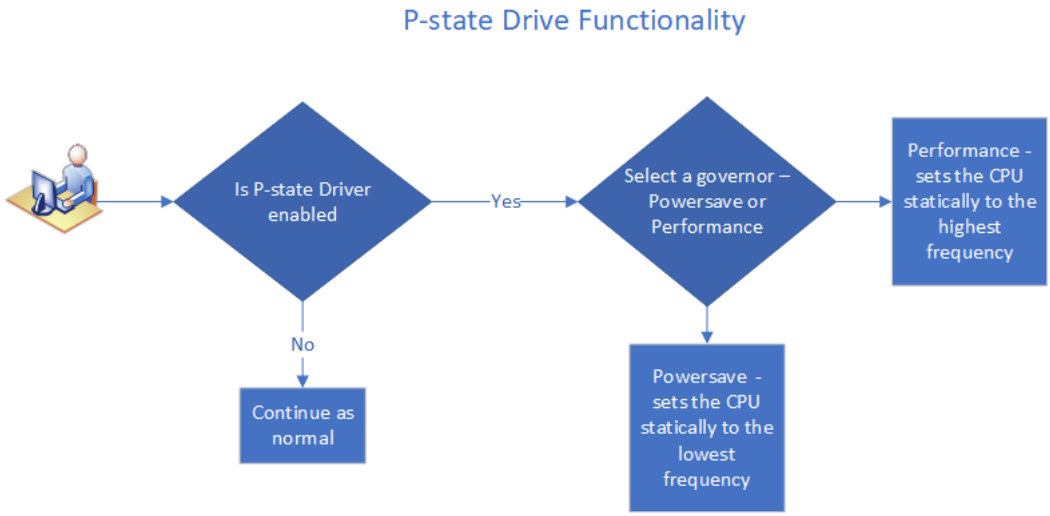

Figure 5. P-state Drive Functionality Flowchart

4.6 Time of Day Functionality

Time of Day is designed to allow the user to select a specific time of day that they can put all their unused CPUs into "sleep" state and then another time to return to "active" state.

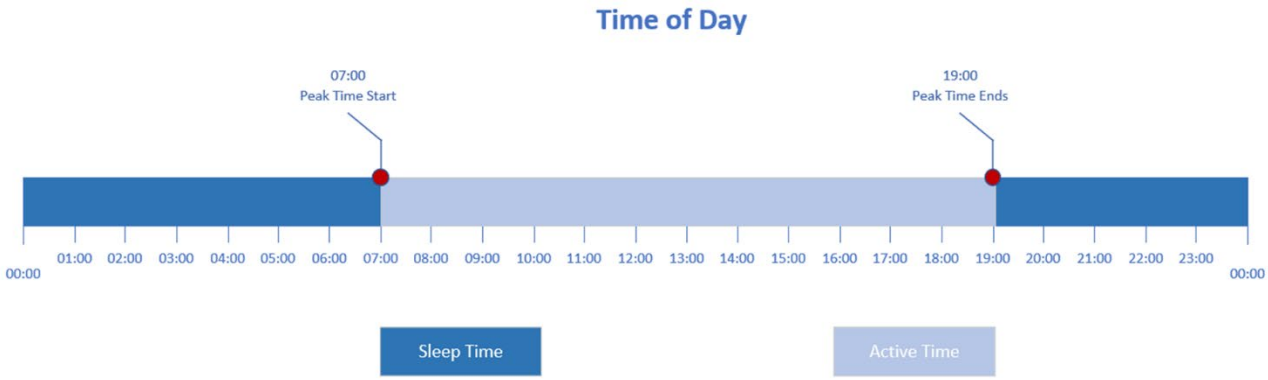

Figure 6. Example Time of Day Functionality

4.6.1 CR Example

The CR example in Figure 7 is called peak-time. In this configuration, it is possible to configure the sleep and active time for the namespace. The outcome of this CR is that the system puts the cores into active state on weekdays at 7:00 AM and into a sleep state at 7:00 PM.

```
apiVersion: time-of-day/v1
kind: peak-time
metadata:
 name: peak-time
spec:
 weekdays: "1-5"
 sleep: "19:00"
 active: "07:00"
 timeZone: "Europe/Ireland"
```

Figure 7. CRD Example

5 Deployments

The Kubernetes Power Manager is responsible for the following tasks.[2]

- Deploying the Power Node agent DaemonSets
- Managing all associated custom resources
- Discovering and advertising PowerProfile extended resources
- Setting up the Kubernetes Power Manager

Use the following steps to set up the Kubernetes Power Manager.

1. Clone Kubernetes Power Manager from https://github.com/intel/kubernetes-power-manager.
2. CD into `kubernetes-power-manager`.
3. Apply the namespace and the role-based access control (RBAC) rules for the operator.

   Run the command: `kubectl apply -f config/rbac/namespace.yaml`

   This command creates the namespace for the Kubernetes Power Manager.

```
apiVersion: v1
```

[2] For workloads and configurations visit Performance Index. Results may vary.

```
kind: Namespace
metadata:
  labels:
    control-plane: controller-manager
  name: intel-power
```

4. Run the command: `kubectl apply -f config/rbac/rbac.yaml`

   This creates the following RBAC rules for the Kubernetes Power Manager service account.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: intel-power-operator
  namespace: intel-power


---


apiVersion: v1
kind: ServiceAccount
metadata:
  name: intel-power-node-agent
  namespace:  intel-power


---


apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: operator-custom-resource-definitions-role
  namespace: intel-power
rules:
- apiGroups: ["", "power.intel.com", "apps", "coordination.k8s.io"]
  resources: ["powerconfigs", "powerconfigs/status", "powerprofiles", "powerprofiles/status",
"events", "daemonsets", "configmaps", "configmaps/status", "leases"]
  verbs: ["*"]


---


apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: operator-custom-resource-definitions-role-binding
  namespace: intel-power
subjects:
- kind: ServiceAccount
  name: intel-power-operator
  namespace: intel-power
roleRef:
  kind: Role
  name: operator-custom-resource-definitions-role
  apiGroup: rbac.authorization.k8s.io


---


apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: operator-nodes
rules:
- apiGroups: ["", "power.intel.com", "apps"]
```

```
  resources: ["nodes", "nodes/status", "configmaps", "configmaps/status", "powerconfigs",
"powerconfigs/status", "powerprofiles", "powerprofiles/status", "powerworkloads",
"powerworkloads/status", "powernodes", "powernodes/status", "events", "daemonsets"]
  verbs: ["*"]


---


apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: operator-nodes-binding
subjects:
- kind: ServiceAccount
  name: intel-power-operator
  namespace: intel-power
roleRef:
  kind: ClusterRole
  name: operator-nodes
  apiGroup: rbac.authorization.k8s.io


---


apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: node-agent-cluster-resources
rules:
- apiGroups: ["", "power.intel.com"]
  resources: ["nodes", "nodes/status", "pods", "pods/status", "powerprofiles",
"powerprofiles/status", "powerworkloads", "powerworkloads/status", "powernodes",
"powernodes/status", "cstates", "cstates/status"]
  verbs: ["*"]


---


apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: node-agent-cluster-resources-binding
subjects:
- kind: ServiceAccount
  name: intel-power-node-agent
  namespace: intel-power
roleRef:
  kind: ClusterRole
  name: node-agent-cluster-resources
  apiGroup: rbac.authorization.k8s.io


---
```

The above commands allow a pod to run with all the CRDs as a service account. They allow the service account to read `configmaps` and update the status.

5. Generate the CRD templates, create the Custom Resource Definitions, and install the CRDs:

```
make
```

Note: Docker images will be pulled from Intel's public Docker Hub and are labeled `intel/power-operator:tag` and `intel/power-node-agent:tag`.

5.1 Running the Kubernetes Power Manager

The following describes how to set up and deploy the Kubernetes Power Manager.

5.1.1 Apply the Manager

Run the command: `kubectl apply -f config/manager/manager.yaml`

The following code creates the `controller-manager-xxxx-xxxx` pod.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: controller-manager
  namespace: intel-power
  labels:
    control-plane: controller-manager
spec:
  selector:
    matchLabels:
      control-plane: controller-manager
  replicas: 1
  template:
    metadata:
      labels:
        control-plane: controller-manager
    spec:
      serviceAccountName:  intel-power-operator
      containers:
      - command:
        - /manager
        args:
        - --enable-leader-election
        imagePullPolicy: Always
        image: intel/power-operator:TAG
         securityContext:
           allowPrivilegeEscalation: false
           capabilities:
             drop: [“ALL”]
        name: manager
        resources:
          limits:
            cpu: 100m
            memory: 30Mi
          requests:
            cpu: 100m
            memory: 20Mi
        volumeMounts:
        - mountPath: /sys/fs
          name: cgroup
          mountPropagation: HostToContainer
          readOnly: true
      terminationGracePeriodSeconds: 10
      volumes:
      - name: cgroup
        hostPath:
          path: /sys/fs
```

5.1.2 PowerConfig

Run the command: `kubectl apply -f examples/powerconfig.yaml`

This command runs the following code and creates the `power-node-agent` DaemonSet that manages the Power Node Agent pods.

```
apiVersion: power.intel.com/v1
kind: PowerConfig
metadata:
  name: power-config
  namespace: intel-power
spec:
  powerNodeSelector:
    # Add labels here for the Nodes you want the PowerNodeAgent to be applied to
    feature.node.kubernetes.io/power-node: "true"
  powerProfiles:
```

```
  # Add wanted PowerProfiles here; valid entries are as follows:
  # performance
  # balance-performance
  # balance-power
  - "performance"
  - "balance-performance"
```

After PowerConfig is deployed, the controller-manager pod sees it via the PowerConfig Controller and creates a Node Agent instance on nodes specified with the `feature.node.kubernetes.io/power-node: "true"` label.

5.1.3 Shared PowerProfile

The example shared PowerProfile in `examples/example-shared-profile.yaml` contains the following PowerProfile spec.[3]

```
apiVersion: power.intel.com/v1
kind: PowerProfile
metadata:
  name: shared
  Namespace: intel-power
spec:
  name: "shared"
  max: 1000
  min: 1000
  epp: "power"
```

Run the following to apply the PowerProfile.

```
kubectl apply -f examples/example-shared-profile.yaml
```

For the PowerProfile controller to recognize a PowerProfile as Shared, its `epp` value must be set to `power`.

5.1.4 Shared PowerWorkload

The example shared PowerWorkload in `examples/example-shared-workload.yaml` contains the following PowerWorkload spec.

Run the command: `kubectl apply -f examples/example-shared-workload.yaml`

This command runs the following code.

Note: Replace all placeholder values with your corresponding cluster values.

```
apiVersion: power.intel.com/v1alpha1
kind: PowerWorkload
metadata:
  name: shared- <NODE_NAME>- workload
  Namespace: intel-power
spec:
  name: "shared- <NODE_NAME>- workload"
  allCores: true
  reservedCPUs:
  - 0
  - 1
  powerNodeSelector:
     kubernetes.io/hostname: <NODE_NAME>
  powerProfile: "shared"
```

The shared workload is the only workload that must be manually created. The shared PowerWorkload created by the user is determined by the PowerWorkload controller to be the designated shared based on the value of `allCores` in the spec. The reserved CPUs on the node must also be specified, as these are not considered for frequency tuning by the controller since they are always being used by Kubernetes' processes. It is important that the value of `reservedCPUs` directly corresponds to the value of `reservedCPUs` in the user's Kubelet config to keep them consistent. The user determines the node for this PowerWorkload using the `PowerNodeSelector` to match the labels on the node. The user then specifies the requested PowerProfile to use.

After the shared workload is created, the PowerWorkload controller creates the corresponding pool in the Intel Power Optimization Library. Then all cores on the system except for the `reservedCPUs` are brought down to this lower frequency level.

[3] For workloads and configurations visit Performance Index. Results may vary.

A shared PowerWorkload that does not begin with `shared-` is rejected and deleted by the PowerWorkload controller. The shared PowerWorkload `powerNodeSelector` must also select a unique node, so it is recommended that the `kubernetes.io/hostname` label be used. A shared PowerProfile can be used for multiple shared PowerWorkloads.

If a shared PowerWorkload is created while another shared PowerWorkload exists on that specific Node, the newly created shared PowerWorkload will be rejected and deleted by the PowerWorkload controller. The shared PowerWorkload's `powerNodeSelector` must also be a unique Node, and it is recommended that you use the `kubernetes.io/hostname` in its configuration. A single shared PowerProfile can be used for multiple Shared PowerWorkloads.

5.1.5 Performance Pod

The example pod in `examples/example-pod.yaml` contains the following PodSpec:

```
apiVersion: v1
kind: Pod
metadata:
  name:  example-power-pod
spec:
  containers:
  - name:  example-power-container
    image: ubuntu
    command: ["/bin/sh"]
    args: ["-c", "sleep 15000"]
    resources:
      requests:
        memory: "200Mi"
        cpu: "4"
        # Replace <POWER_PROFILE> with the PowerProfile you wish to request
        # IMPORTANT: The number of requested PowerProfiles must match the number of requested CPUs
        # IMPORTANT: If they do not match, the Pod will be successfully scheduled, but the
PowerWorkload for the Pod will not be created
        power.intel.com/<POWER_PROFILE>: "4"
      limits:
        memory: "200Mi"
        cpu: "4"
        # Replace <POWER_PROFILE> with the PowerProfile you wish to request
        # IMPORTANT: The number of requested PowerProfiles must match the number of requested CPUs
        # IMPORTANT: If they do not match, the Pod will be successfully scheduled, but the
PowerWorkload for the Pod will not be created
        power.intel.com/<POWER_PROFILE>: "4"
```

Replace the placeholder values with the PowerProfile you require and apply the PodSpec:

```
kubectl apply -f  examples/example-pod.yaml
```

At this point, if only the performance PowerProfile was selected in the PowerConfig, the user's cluster contains three PowerProfiles and two PowerWorkloads.

```
kubectl get powerprofiles -n intel-power
```

Example result:

```
NAME                          AGE
performance                   59m
performance-<NODE_NAME>       58m
shared-<NODE_NAME>            60m
```

```
kubectl get powerworkloads -n intel-power
```

Example result:

```
NAME                                AGE
performance-<NODE_NAME>-workload    63m
shared-<NODE_NAME>-workload         61m
```

5.1.6 Delete Pods

Run the following to check for pods and delete them.

```
kubectl get pods
kubectl delete pods <name>
```

The PowerWorkload associated with the pod is deleted. If other pods are using it, it is updated and not deleted.

The Intel Power Optimization Library instance pool for the associated workload is also deleted. The cores that were returned from that pool are returned to the shared frequencies.

5.2 Extended Resources

PowerProfiles are advertised as extended resources on all nodes that match the `PowerConfig.PowerNodeSelector` labels.

The extended resources give the Kubelet control over the number of PowerProfiles a given frequency level can have, making sure that the board is not overcommitted with higher frequency cores that could potentially overheat it.

The number of extended resources is determined by the name of the PowerProfile, of which there can be four. Each PowerProfile can have a percentage of the overall number of cores on the node, as follows.

- Performance – 40%
- Balance-performance – 60%
- Balance-power – 80%
- Power – 100%

For example, an Intel SST-capable node with a capacity of 72 CPU resources advertises 28 `sst.intel.com/performance` resources (72 x 40% = 28):

```
Capacity:
  cpu:                          72
  power.intel.com/performance:  28
Allocatable:
  cpu:                          70
  power.intel.com/performance:  28
```

This is an example of a node's resources, requestable via the PodSpec's container resource requests.

It is essential that you request CPU and `sst.intel.com/performance` resources in equal amounts.

5.3 Recommended Approach for Use with CPU Manager

It is recommended that nodes to be governed by the Kubernetes Power Manager (i.e., nodes that match the `powerNodeSelector` labels in the PowerConfig) be designated exclusively to containers that require CPUs with optimized Intel SST settings.

After these nodes are labeled and tainted as such, the operator can work harmoniously with the node's CPU Manager to (a) configure the shared pool with a power-saving PowerProfile, and (b) configure exclusively allocated CPUs with performance-enhancing PowerProfiles as requested.

Intel SST capabilities must be enabled on the node for Kubernetes to expose the capabilities. Without it enabled, PowerProfiles in the Kubernetes Power Manager cannot function correctly.

6 Result

6.1 Shared Pool

The reserved CPUs on all `intel.power.node` nodes continue to run with the default configuration and are not impacted by any PowerProfile. The allocatable CPUs (shared pool – reserved CPUs) on all `intel.power.node` nodes are configured with the standard performance PowerProfile.[4]

6.2 Exclusive Allocated CPUs

The high performance pod is scheduled to a designated `intel.power.node` node. The high-performance pod's container is allocated three exclusive CPUs by the CPU Manager. The Kubernetes Power Manager configures these three CPUs with the settings of the high-performance PowerProfile.

7 Summary

Kubernetes operators are used extensively within the open-source community. The Kubernetes Power Manager uses Intel-specific power management technologies. This reduces the number of CPUs active on a system at any given time, which then reduces the power consumption. By reducing power consumption, the user helps the environment and reduces operating and overhead costs.

This benefits a user greatly when the time affects the load on their server, ramping the frequencies of cores down during off-peak hours and powering them up when high-priority workloads are scheduled. This can also be hugely beneficial for NFV

[4] For workloads and configurations visit Performance Index. Results may vary.

infrastructure containers, for example Open vSwitch as it ensures that the high-priority containers running the packet processing activities run on cores that are configured for optimal frequencies, guaranteeing a level of performance.

1022/DN/WIPRO/PDF 647910-003US