USER GUIDE

Intel Corporation

Resource Management Daemon

Authors

Joseph Gasparakis

Dakshina Ilangovan

Jasvinder Singh

1 Introduction

Increasing the number of CPU cores and storage capacity in data centers has greatly increased the density of software components per server. This high density causes software components to compete for finite resources on the platform, possibly producing a negative impact on the performance, service assurance, and the end user's general quality of experience. The Resource Management Daemon (RMD) is an open source software component that arbitrates between resources and software components.

RMD runs on the server and maps resources (or pools of them) to software components according to a certain policy. For each workload, RMD exposes a RESTful API to receive a policy from external management software (such as orchestrator, container engine, customer management plane, and so forth) and enforces that policy on the platform.

RMD handles arbitration locally on the compute node (rather than sending telemetry to the upper layers of the stack to make the decision there). The reason is that some use cases, such as latency sensitive applications, require faster local analysis and mitigation of resource contention.

This document describes RMD architecture and installation, provides a usage scenario, discusses plugins and OpenStack* integration, and provides an example of using RMD for the Intel® Resource Director Technology (Intel® RDT) and P-state control using Performance Monitoring Unit (PMU).

- RMD enables an abstracted and scalable implementation for finer grain resource management.
- RMD can manage a subset of platform resources through support for multiple interfaces on the host platform.
- RMD provides a centralized and holistic decision-making mechanism for resource management that can be supported on multiple platform types.
- RMD can be integrated with any node management layer that supports node or workload policies.

This document is part of the Network Transformation Experience Kit, which is available at: https://networkbuilders.intel.com/network-technologies/network-transformation-exp-kits

Table of Contents

Figures

Tables

1.1 Terminology

Table 1. Terminology

Abbreviation	Description
CAT	Cache Allocation Technology
CLOS	Class of Service
LLC	Last Level Cache
NFV	Network Functions Virtualization
PAM	Pluggable Authentication Modules
PMU	Performance Monitoring Unit
QoS	Quality of Service
RDT	Intel® Resource Director Technology (Intel® RDT)
RDT-CAT	Intel® Resource Director Technology – Cache Allocation Technology (Intel® RDT-CAT)
RMD	Resource Management Daemon
RPC	Remote Procedure Calls
VNF	Virtual Network Function
VNF-M	Virtual Network Function Manager
vRouter	Virtual Router
vSwitch	Virtual Switch

1.2 Reference Documents

Table 2. Reference Documents

Reference	Source
Intel® Resource Director Technology	https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html
Resource Management Daemon repository on GitHub*	https://github.com/intel/rmd
Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family	https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology
Introduction to Memory Bandwidth Monitoring in the Intel® Xeon® Processor E5 v4 Family	https://software.intel.com/en-us/articles/introduction-to-memory-bandwidth-monitoring

2 Architecture

RMD is a daemon that runs on the server, one instance per compute node. It exposes a RESTful API for other software layers to make requests and push policies that perform mapping between workloads and resources. These layers need to authenticate themselves through Pluggable Authentication Modules (PAM). There is also HTTPS support accessing the REST API.

The policy can be static or dynamic:

- Static policy is a one off allocation of resources. RMD allocates these resources to the associated workload and then ensures that the workload is still running. If the workload stops running, RMD releases the resources so that they are available for consumption by the next workload that is scheduled on the server.
- When a dynamic policy is received, RMD constantly monitors and re-allocates resources as described in the policy. One can describe an algorithm in that policy on how RMD should do this. As an example, for a multi-way cache, the policy might request for the workload to receive two cache ways from the processor's last level cache (LLC), but to allocate an extra cache way if the throughput on a specific network interface increases above a certain threshold, or to release the extra cache way if the throughput decreases below the threshold.

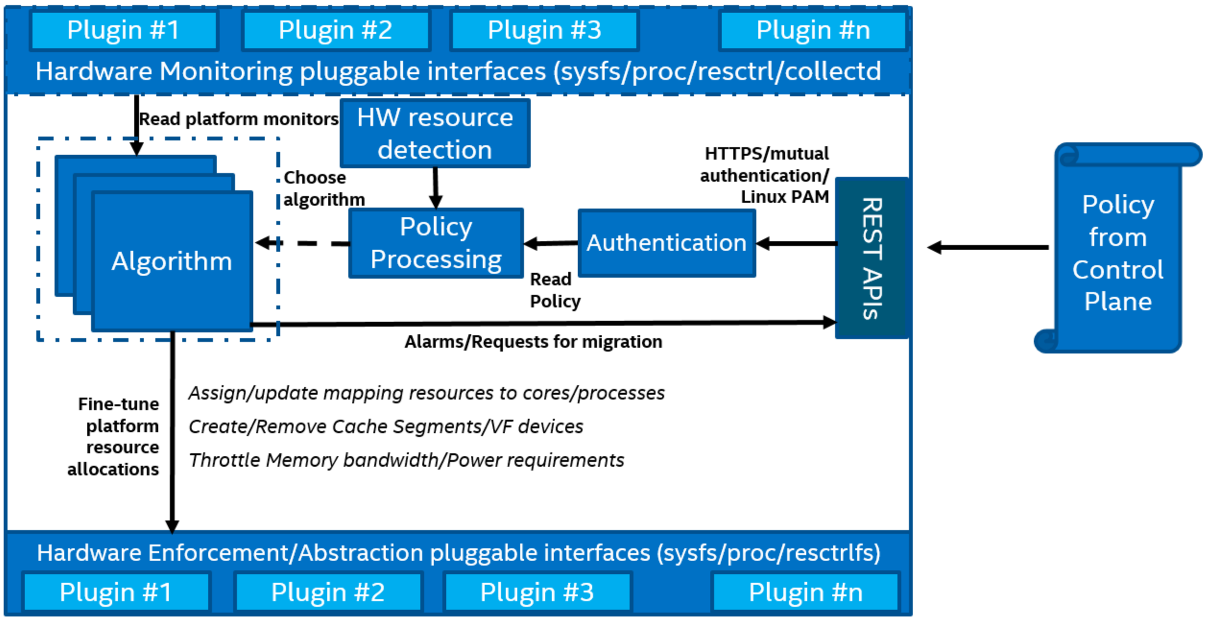

Plugin #1
Plugin #2
Plugin #3
Plugin #n
Hardware Monitoring pluggable interfaces (sysfs/proc/resctrl/collectd
Read platform monitors
HW resource detection
Choose algorithm
Algorithm
Policy Processing
Authentication
HTTPS/mutual authentication/ Linux PAM
REST APIs
Policy from Control Plane
Read Policy
Alarms/Requests for migration
Fine-tune platform resource allocations
Assign/update mapping resources to cores/processes
Create/Remove Cache Segments/VF devices
Throttle Memory bandwidth/Power requirements
Hardware Enforcement/Abstraction pluggable interfaces (sysfs/proc/resctrlfs)
Plugin #1
Plugin #2
Plugin #3
Plugin #n

Figure 1. RMD Internal Architecture

Figure 1 shows the internal architecture of RMD. When a policy is received by an authenticated software entity, it is processed and validated against a hardware resource detection mechanism (if the resources exist, are available, and RMD can control them.). Provided everything is valid, RMD spawns a thread that works in a loop, monitoring resources constantly, and re-allocating them as needed and if the policy is of dynamic nature.

Monitoring and enforcement (re-allocation) of resources is done through pluggable interfaces. For RMD to monitor and control a new resource, you must implement the right plugin and RMD will be able to support this resource onward. For more details, see Section 5.6.

Figure 2 below shows that structurally, RMD is broken into user and root engines. The user engine provides the RESTful API, runs the monitoring threads, and runs the re-enforcing threads. The engine with elevated privileges (Super user process) performs the actual monitoring and allocations and uses Remote Procedure Calls (RPC) to communicate the information to and from the user engine. Pluggable Authentication Modules (PAM) authentication is also handled by the root process.

RMD uses NoSQL databases for persisting policy information, shown below in Figure 2 as Bolt and Mongo* databases. If RMD terminates abnormally, it reads the previously stored database information and restores the resource allocations. It also uses configuration files in order to organize resources at start up.

Note: Blocks with a dotted border in Figure 2 are used to indicate that monitor threads, dynamic allocation algorithms, resource plugins, and the Redis* data store are planned for future development.

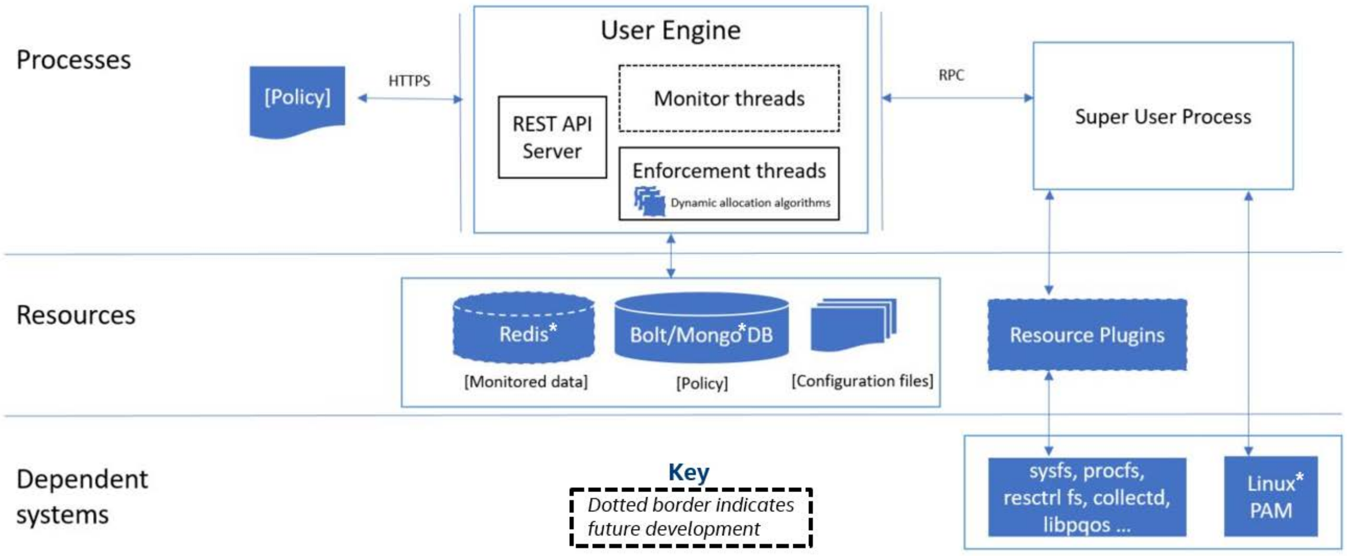


Figure 2. RMD System Level Architecture

2.1 RMD Implementation Examples

This section describes the following RMD implementation examples:

- management of allocations to Last Level Cache (LLC) using Intel® RDT-CAT (Intel® Resource Director Technology – Cache Allocation Technology)
- management of P-state using PMU counters

2.1.1 RMD for Static Allocation of Last Level Cache (LLC)

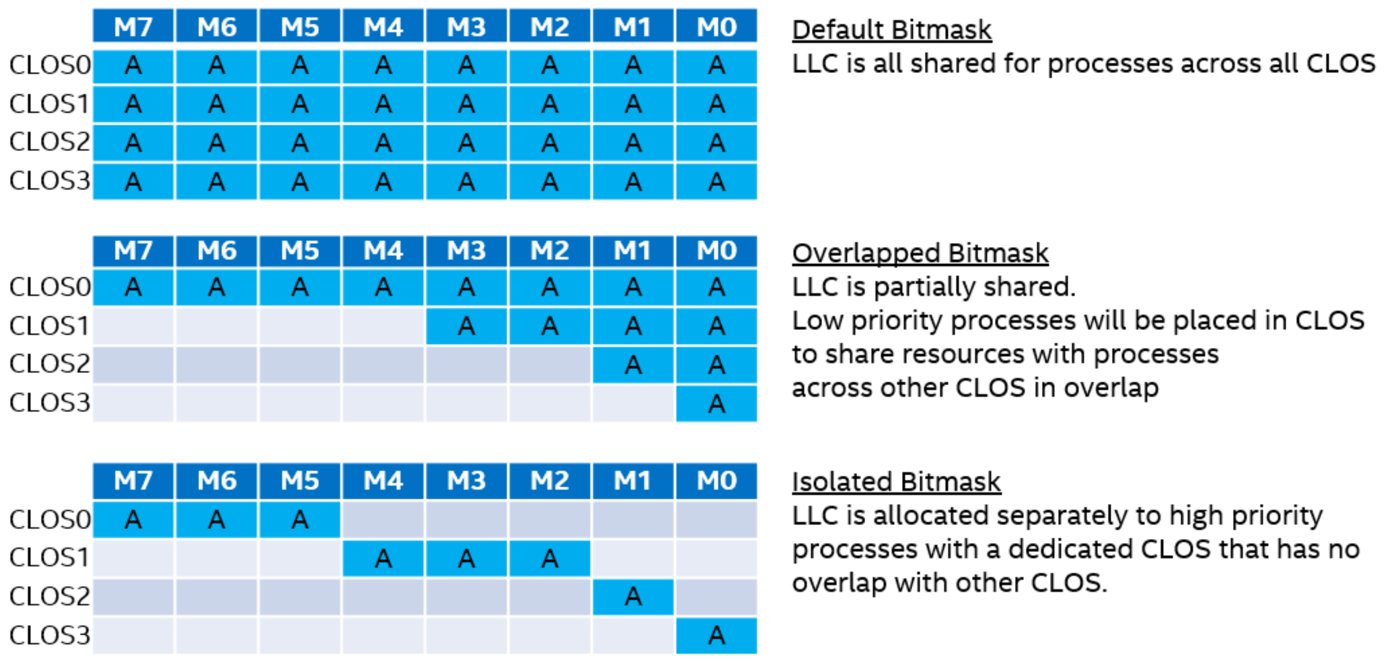
RMD controls the Last Level Cache on a host using Cache Allocation Technology (CAT) from Intel® Resource Director Technology (Intel® RDT). For more information on Intel® RDT, see https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html

CAT introduces the concept of describing multiple LLC available on a system. On current generation systems there is one L3 cache per socket but CAT allows for multiple separate L3 caches, representing each LLC using a Cache ID.

Each LLC is measured in cache ways. For example, a four-way cache has four cache ways. The size of a cache way is the total size of the LLC divided by the number of cache ways available. The cache ways are logically partitioned using Class of Service (CLOS). This segment of LLC is assigned for sole use by one or more processes or cores. The number of available CLOS is platform dependent.

Currently the allocation of cache ways to processes or cores with CLOS can be controlled using MSR registers or Linux* `resctrl` file system starting kernel version 4.10.

Figure 3 illustrates a logical grouping of cache ways into CLOS that determine how cache ways are shared.



	M7	M6	M5	M4	M3	M2	M1	M0
CLOS0	A	A	A	A	A	A	A	A
CLOS1	A	A	A	A	A	A	A	A
CLOS2	A	A	A	A	A	A	A	A
CLOS3	A	A	A	A	A	A	A	A

Default Bitmask
LLC is all shared for processes across all CLOS

	M7	M6	M5	M4	M3	M2	M1	M0
CLOS0	A	A	A	A	A	A	A	A
CLOS1					A	A	A	A
CLOS2							A	A
CLOS3								A

Overlapped Bitmask
LLC is partially shared.
Low priority processes will be placed in CLOS to share resources with processes across other CLOS in overlap

	M7	M6	M5	M4	M3	M2	M1	M0
CLOS0	A	A	A					
CLOS1				A	A	A		
CLOS2							A	
CLOS3								A

Isolated Bitmask
LLC is allocated separately to high priority processes with a dedicated CLOS that has no overlap with other CLOS.

Figure 3. Logical Grouping of Cache Ways into CLOS

RMD provides hooks into the resctrl file system to further associate Quality of Service (QoS) meta-data to Intel® RDT CLOS that can be provided. With features to create CLOS for processes such that cache ways can be allocated in dedicated, shared or overlapped manner, RMD can allocate cache ways for processes into three QoS based cache pools. Refer to Figure 4.

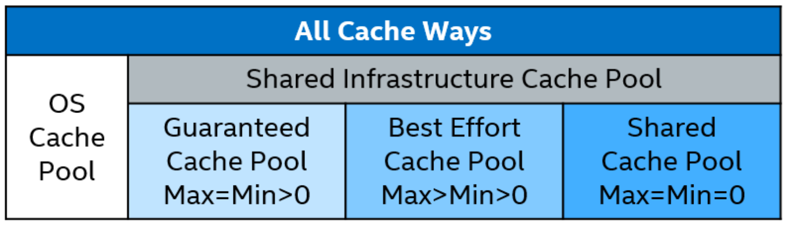

All Cache Ways
OS Cache Pool
Shared Infrastructure Cache Pool
Guaranteed Cache Pool Max=Min>0
Best Effort Cache Pool Max>Min>0
Shared Cache Pool Max=Min=0

Figure 4. RMD Partitioning of Cache Ways into Cache Pools using Intel® RDT CLOS Constructs

The **Guaranteed** cache pool allocates dedicated cache ways for all processes requesting from it. This request is provided to RMD as `Max_Cache=Min_Cache, Max_Cache>0`, and `Min_Cache>0`.

The **Best Effort** cache pool allocates cache ways within the window of `Min_Cache` and `Max_Cache` described where `Max_Cache>0` and `Min_Cache>0`. RMD attempts to allocate `Max_Cache` upon request and can oversubscribe processes on host by changing allocations based on usage. This cache pool is leveraged with dynamic re-allocations.

The **Shared** cache pool allocates all processes to a fixed size CLOS (set of cache ways). All processes in this pool share their cache ways. Low priority/short-lived processes can be subscribed to this pool. The number of processes in this pool is configurable in RMD to avoid any contention.

Two other statically configured cache pools are provided:

- The **OS** cache pool is dedicated for operating system processes.
- The **Shared Infrastructure** cache pool is configured in overlap with all available cache pools except the OS cache pool. This enables data sharing between infrastructure processes subscribed to other processes that can be scheduled on other cache pools. A typical example would be for a network functions virtualization (NFV) use case with virtual switch (vSwitch) or virtual router (vRouter) processes that require sharing of data in LLC with other virtual network functions (VNFs).

2.1.2 RMD for P-State Control of CPU Cores

In some NFV deployments, we may be limited to observing CPU metrics alone to achieve a power saving mode. Consider the following:

- Networking workloads are typically running in a polling mode. Observing the CPU utilization in polling mode always shows 100% busy.
- For NFV, the workloads may also be virtualized and direct control of P-states is not typically facilitated through Guest OS kernels.
- Application telemetry may not be available to observe the real utilization of the application.

The deployment model for this plugin is to deploy to a host OS and control the frequency of cores on behalf of a Virtual Machine, without any feedback loop from the VM.

When the deployment is limited by the above criteria, it is possible to detect idle or busy periods by viewing the CPU metrics. This detection mechanism is limited to idle or busy and not how busy, as this requires characterization per workloads.

To achieve this, the scheme observes CPU counters to detect a change in behavior to idle or the code is busy doing work. When idle is detected, the frequency can be scaled down and when work is detected, the frequency can be scaled up.

Per-core P-states is a CPU feature that can be used to save power by scaling frequency. For 100% utilization or polling workloads, a trigger is required to achieve a power saving. This mechanism allows detection of a change of condition when still 100% utilized and is the trigger that can be used to make a power saving decision.

RMD interfaces with the P-state control plugin that helps minimize platform power consumption by exploiting the temporal variation in traffic volume. The plugin targets power saving for virtualized application with no feedback loop to the host and those workloads are 100% polling software threads. The plugin monitors the x86 branch instruction misses to hit ratio in the Performance Management Unit (PMU) of the CPU cores to determine the traffic load.

The ratio of branch misses to branch hits grows significantly when the traffic flow increases, since the number of instructions to execute increases while processing the packets. When there are no packets to process, the branch miss to hit ratio is very low, in percentage terms, typically <0.01%. At 100% traffic, the branch miss to hit ratio is notably higher, in the order of 2%.

PMU counters such as the branch miss to hit ratio of a core suggests its utilization, which the software can then assess to increase or decrease CPU frequency (that is, decrease frequency when utilization is low, for example) without modifying the application. The tradeoffs are that a lower CPU frequency can yield power savings, whereas a higher frequency can improve performance.

As shown in Figure 5, on receiving workload policy requests, the RMD core performs a sanity check and when passed, forwards requests to the P-state plugin that runs out of band on a dedicated core. During runtime, the plugin continuously inspects the branch misses provided by the PMU counters of the CPU cores running the workloads, compares them with the threshold value from the RMD workload policy to determine the workload utilization (in this case, the traffic rate), and then scales up or scales down the CPU core frequency accordingly to increase performance or save power, respectively.

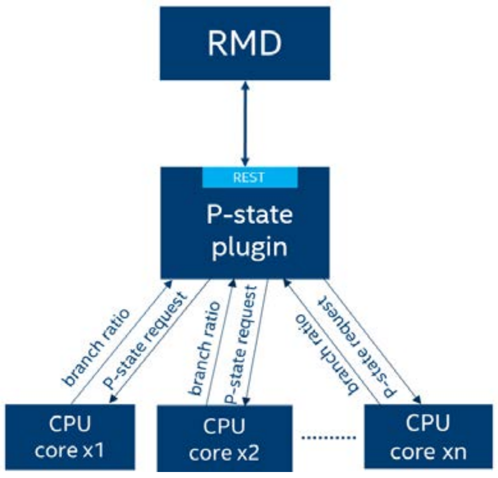


Figure 5. CPU P-State Control Plugin

3 Integration with Orchestration Layers

Note: This section describes a planned implementation, which will be released at a future date.

From an orchestration perspective, interaction with a physical host is limited to the initial allocation of resources and the characterization of nodes to improve the placement of workloads. Runtime handling of workload requirements is not part of the core functionality.

The future implementation of RMD acting as an external component to orchestration environments like OpenStack* or Kubernetes* will offer the functionality to manage different node capabilities and will provide configurability to specify the policies and changes required by the sys-admin, the tenant or the workload deployment requirements. It will provide a unified API for the resource orchestrator to enable high frequency and autonomic control to the node within a set of guardrails defined by a node policy to improve the workload functional and non-functional characteristics.

3.1 RMD Integration with OpenStack*

Note: This section describes a planned implementation, which will be released at a future date.

Nova is the compute component in OpenStack*. It is responsible to provide awareness to OpenStack on the compute node capabilities and also provisions and manages the lifecycle of workloads (VNFs) scheduled on it. RMD, as a component on each compute node external to OpenStack services, will take care of validation and enforcement of policies provisioned by two different roles: the Sys-admin and the VNF Manager (VNF-M). The compute node will be provisioned with a set of RMD resource plugins to support the Sys-admin and the VNF-M requests.

The Sys-admin will provide the “Node Policy” to enable a subset of host resources and supply node level configuration for those resources. The VNF-M will supply “Workload Policy” along with VNF requirements. The “Workload Policy” will be used to describe the resources that can be provisioned or controlled by RMD during the lifecycle of the VNF. The RMD component will be in charge of verifying the applicability of the “Node Policy” and will also deliver to Nova the resource characteristics enabled on the node before Nova services are initiated.

Once the Nova services are started, the VNF-M will be able to request the VNF requirements along with the “Workload Policy” for the VNF. The VNF requirements will be used for scheduling the VNF on a compute node with matching specs.

Upon creation of the VNF on a compute node with the VNF requirements, the RMD component on that compute node will asynchronously execute on the “Workload Policy” it queries from Nova in an out-of-band manner. The RMD component will apply the specific changes to the hardware using RMD resource plugins.

4 Installation Guide

This section contains RMD instantiation instructions. RMD is currently supported as a system daemon running on generic Linux platforms/x86 platforms. RMD supports the static allocation and QoS management of an LLC hardware resource using Intel® RDT – CAT technology and CPU P-state control using PMU counters.

4.1 Prerequisites

The prerequisites for deploying RMD are as follows:

- RMD is supported on select x86 CPU SKUs that enable Intel® RDT – CAT.
- RMD is supported on select Linux* kernel versions (4.10 or greater) that enable `resctrl fs`. For details, refer to https://www.kernel.org/doc/Documentation/x86/intel_rdt_ui.txt

1. To enable `resctrl fs` with Intel® RDT – CAT support, add the following to the GRUB command line:
   ```
   rdt=l3cat
   ```
2. To verify support for P-state control, check for the scaling driver using the command:
   ```
   cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_driver | egrep “acpi_cpufreq|intel_pstate”
   ```
3. To verify minimum and maximum frequency limits defined by the manufacturer for a CPU, use the commands:
   ```
   cat /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_min_freq
   cat /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_max_freq
   ```
4. To enable PMU counters, load the model specific register driver for PMU counters using the command:
   ```
   modprobe msr
   ```
5. Verify this on the deployment node using the command:
   ```
   cat /proc/filesystems  | grep resctrl
   ```
6. Mount the `resctrl fs` using the command:
   ```
   mount -t resctrl resctrl /sys/fs/resctrl
   ```
7. Install the Linux PAM, Berkeley DB, and OpenSSL packages.
8. To install the Debian* or Ubuntu* OS, enter the command:
   ```
   sudo apt-get install openssl libpam0g-dev db-util
   ```
9. To install the Red Hat* Linux* OS, enter the command:
   ```
   sudo dnf install openssl pam-devel db4-utils
   ```

4.2 Install as Executable

RMD follows versioned history and with each stable release, an executable will be released on GitHub in the location: https://github.com/intel/rmd/releases

The executable file must be executed with super user privileges and can be added to the system path.

4.3 Install from Source

The RMD source code can be cloned from the GitHub repository: https://github.com/intel/rmd.git

To build RMD from source, the system should support the Golang environment. For instructions, refer to https://golang.org/doc/install and ensure `rmd` repo is cloned to GOPATH `src` directory.

The following commands are a shorthand to get started with building and running RMD:
1. Download RMD source code dependencies and build:
   ```
   make
   ```
2. Install RMD binary to build/ directory:
   ```
   scripts/install.sh
   ```
3. Run rmd from build directory in debug mode:
   ```
   rmd –d
   ```
4. Test RMD REST interfaces exposed over HTTP in debug mode and query LLC information:
   ```
   wget http://localhost:8081/v1/cache/l3
   ```

The above steps will help users and developers to quickly get started with building and running RMD using default configuration files. For detailed documentation, refer to the links below.
- Configuration guide: https://github.com/intel/rmd/blob/master/docs/ConfigurationGuide.md
- User guide: https://github.com/intel/rmd/blob/master/docs/UserGuide.md
- Developer guide: https://github.com/intel/rmd/blob/master/docs/DeveloperQuickStart.md

5 Usage Examples

5.1 CAT – Node Policy

In order to associate cache ways to workloads efficiently using RMD, the host should be prepared by applying “Node Policy” using the RMD configuration file.

The RMD configuration file supports fields to specify cache allocation groups and quantities for reserved processes (such as operating system) and infrastructure processes in general.

The default configuration file is installed at `/usr/local/etc/rmd/rmd.toml.`
```
[OSGroup] # mandatory
cacheways = 1
cpuset = "0-1"

[InfraGroup] # optional
cacheways = 19
cpuset = "2-3"
# arrary or comma-separated values
tasks = ["ovs*"] # just support Wildcards
```

```
[CachePool]
shrink = false # whether allow to shrink cache ways in best effort pool
max_allowed_shared = 10 # max allowed workload in shared pool, default is 10
guarantee = 10
besteffort = 7
shared = 2
```

The configuration file directives are explained below:

- **OSGroup**: cache ways reserved for use by the operating system usage.
- **InfraGroup**: cache ways dedicated for use by identified infrastructure process. These cache ways are allocated to be shared with every other workload scheduled on the system.
- **CachePool**: Cache allocation requests are grouped into one of three groups:
  - **guarantee**: allocate dedicated cache ways to a workload (`max_cache == min_cache > 0`)
  - **besteffort**: dedicated minimum number of cache ways for workload which can grow to maximum number of cache ways in an overlapped manner (`max_cache > min_cache > 0`)
  - **shared**: all workloads in this pool share cache ways (`max_cache == min_cache = 0`)

  Flags are:
  - **shrink**: flag to indicate whether to shrink cache ways already allocated to workloads in **besteffort** pool upon arrival of a new workload.
  - **max_allowed_shared**: threshold to define the max allowed workloads in shared cache pool

Refer to Figure 4 and Section 2.1 for additional information.

On a host that supports up to 20 cache ways, the configuration file will create the following cache bit mask layout.

```
OSGroup:    0000 0000 0000 0000 0001
InfraGroup: 1111 1111 1111 1111 1110
```

The layout of the remaining cache ways available for cache pools represented by the cache bit mask is:

```
guarantee:  0000 0000 0111 1111 1110
besteffort:  0011 1111 1000 0000 0000
shared:       1100 0000 0000 0000 0000
```

5.2 Querying Cache Inventory Information

The LLC information on the node deploying RMD can be queried using the following REST API calls:

```
$ curl -i http://127.0.0.1:8081/v1/cache/
$ curl -i http://127.0.0.1:8081/v1/cache/l3
$ curl -i http://127.0.0.1:8081/v1/cache/l3/0
```

5.3 P-State Control – Node Policy and Configuration

To use RMD to perform P-state control on the CPU cores running the workloads, the host should be prepared to configure the P-state control plugin using the RMD configuration file.

The RMD configuration file contains an optional P-state section for specifying configuration and node policy for the P-state control plugin. If this section is not present in the configuration file, the P-state plugin interface in RMD will be disabled.

```
[pstate] # optional

enabled = true   # Simple plugin enable/disable flag

port = 8080 # port number with plugin's http server (REST API)
```

The configuration file directives are explained below:

- **enabled**: when set to "false" disables the P-state module on the node.
- **port**: defines the listening port for branch_monitor's http server with REST API interface.

5.4 Querying P-State Control Information

The P-state configuration parameters for the CPU cores can be queried using the following REST API calls:

```
$ curl -i http://127.0.0.1:8081/v1/pstate/
$ curl -i http://127.0.0.1:8081/v1/pstate/{core_id}
```

5.5 Workload Policies to Request LLC Allocation

5.5.1 Changing and Adding Policies

The Sys-admin can change and add new policies by editing the following line in the yaml file which is pointed to in the configuration file.

```
policypath = "etc/rmd/policy.toml"
```

5.5.2 Predefined Policy

The Sys-admin can provide predefined LLC allocation policies based on the target platform characteristics. The predefined allocation policy represents cache pools of Guaranteed, Best-effort, and Shared using Gold, Silver, and Bronze levels respectively.

The following example is a predefined policy which can be referenced in workload policy to RMD to request LLC.

Note: The following `catpolicy` file includes product codenames for processors that have been released by Intel.

Cascadelake is branded as 2nd generation Intel® Xeon® Scalable processors.

For desktop and mobile, Skylake is branded as 6th Generation Intel® Core™ i3, Intel® Core™ i5, and Intel® Core™ i7 processors. For workstations, Skylake is branded as Intel® Xeon® E3 v5 processors.

```
catpolicy:
    cascadelake:
        - gold:
            - MaxCache: 4
            - MinCache: 4
        - silver-bf:
            - MaxCache: 2
            - MinCache: 1
        - bronze-shared:
            - MaxCache: 0
            - MinCache: 0
    skylake:
        - gold:
            - MaxCache: 3
            - MinCache: 3
        - silver-bf:
            - MaxCache: 2
            - MinCache: 1
        - bronze-shared:
            - MaxCache: 0
            - MinCache: 0
```

Example Workload Request Command

The following command creates a workload request to RMD with gold policy for a workload described with process id 78377:

```
$ curl -H "Content-Type: application/json" --request POST --data \
         '{"task_ids":["78377"], "policy": "gold"}' \
         http://127.0.0.1:8081/v1/workloads
```

5.5.3 Custom Policy

To specify a custom policy to request LLC allocation, the payload body should contain the following. A workload can be described as a set of task_ids or set of core_ids.

- **task_ids**: set of process ids describing the workload
- **core_ids**: set of core ids on which the workload process ids run
- **max_cache**: maximum number of cache ways the workload can run on
- **min_cache**: minimum number of cache ways the workload must run on

Example workload request command

The following command creates a workload request to RMD with minimum and maximum cache values for a workload described with process id 78377:

```
$ curl -H "Content-Type: application/json" --request POST --data \
         '{"task_ids":["78377"], "max_cache": 4, "min_cache": 4}' \
         http://127.0.0.1:8081/v1/workloads
```

5.5.4 Hospitality Score of a Workload

The hospitality score API provides a score in the range of 0 to 100 to indicate how hospitable the host is to provide the desired LLC allocation values for the workload. The score is calculated as shown in Table 3.

Table 3. Hospitality Score Calculation

Request	Hospitality Score	LLC Pool
max_cache == min_cache > 0	[0 | 100]	Guarantee
max_cache == min_cache == 0	[0 | 100]	Shared
max_cache > min_cache > 0	[0 , 100]	Best-effort

Example: The following code will display the hospitality score for a workload that has desired LLC allocation values of min=2 and max=2. Since two LLC ways can be provided on the host, the score is returned as 100.

```
$ curl -H "Content-Type: application/json" --request POST --data \
         '{"max_cache": 2, "min_cache": 2}' \
         http://127.0.0.1:8888/v1/hospitality
{
    "score": {
        "l3": {
            "0": 100,
            "1": 100
        }
    }
}
```

5.6 Workload Policies to Request CPU P-State Control

To specify a workload policy to request P-state control for the workload running on CPU core, the payload body should contain the following:

- **core_ids**: the set of core ids on which the workload process ids run
- **pstate_br**: branch miss threshold value
- **monitoring**: flag to enable or disable P-state control for the workload

Example workload request command

The following command creates a workload request to RMD to enable P-state control for CPU core 7.

```
$ curl -H "Content-Type: application/json" --request POST --data \
         '{"core_ids":["7"], "pstate_br": 3.0, "monitoring": "on"}' \
         http://127.0.0.1:8081/v1/workloads
```

6 Plugins

RMD is a centralized arbitrator to perform resource management on the platform. For this it needs access to system resources for enforcement as well as monitoring. For ease of use for platform and vendor developers, RMD defines standard interfaces for each resource it can control. Platform and vendor developers can write plugins that can conform to these provided standard interfaces. We can expect multiple versions for a single plugin for a resource as well as multiple plugins for a resource. Plugins can be combined to handle both write and read but must be registered separately as write and read plugins with RMD for better management.

Using the plugin approach:

- RMD administrators can control the list of resources supported by RMD for enforcement and monitoring through configuration. It is the responsibility of administrators to ensure the plugins configured are compatible with RMD and the platform. For example, compatibility might be required for Golang, the kernel, hardware versions like CPU model, and so forth.
- Platform and vendor developers should provide template configuration and documentation to explain the features provided by the plugin as well as any dependencies required.
- For a resource, the configuration should allow only one plugin each for write and read operations.

Different resources on the system will have different needs and the configurability and control will be highly coupled to the resource controller implementation. RMD plugins reduce the dependency on platform or vendor specific implementation details. It uses its configuration to load the correct plugin based on platform or vendor resource version, if provided.

Usually the RMD plugins are pieces of code that read or write into standard kernel facilities such as the pseudo filesystems (`proc/sysfs/resctrlfs` and others) but they can be implemented by accessing resources in different ways: through communicating with other daemons on the platform, calling libraries, and writing directly to Model Specific Registers (MSRs).

To support a newly introduced resource, you need to do a bit of research on how this resource is exposed. Starting from the kernel pseudo filesystem is usually a good approach, but there might be other ways that a resource can be accessed, such as a command line tool (such as `sysctl` or `ethtool`) or via another daemon or library.

It's important to experiment to get an understanding of how this resource affects workloads. As an example, refer to this investigation for Cache Allocation Technology here. Then you can proceed with writing a plugin that monitors and enforces the resource in a meaningful way.

7 Summary

Resource Management Daemon (RMD) enables a flexible and scalable implementation for resource management. It provides a unified centralized decision making mechanism that can be used in multiple environments.

This document described RMD architecture and installation, provided a usage scenario, discussed plugins and OpenStack integration, and provided examples of using RMD for the Intel® Resource Director Technology (Intel® RDT) and for P-state control using PMU.