User Interface for Resource Control feature

Authors: Fenghua Yu <fenghua.yu@intel.com>
Tony Luck <tony.luck@intel.com>
Vikas Shivappa <vikas.shivappa@intel.com>

Intel refers to this feature as Intel Resource Director Technology(Intel(R) RDT). AMD refers to this feature as AMD Platform Quality of Service(AMD QoS).

This feature is enabled by the CONFIG_X86_CPU_RESCTRL and the x86 /proc/cpuinfo flag bits:

RDT (Resource Director Technology) Allocation	"rdt_a"
CAT (Cache Allocation Technology)	"cat_l3", "cat_l2"
CDP (Code and Data Prioritization)	"cdp_l3", "cdp_l2"
CQM (Cache QoS Monitoring)	"cqm_llc", "cqm_occup_llc"
MBM (Memory Bandwidth Monitoring)	"cqm_mbm_total", "cqm_mbm_local"
MBA (Memory Bandwidth Allocation)	"mba"

To use the feature mount the file system:

```
# mount -t resctrl resctrl [-o cdp[,cdpl2][,mba_MBps]] /sys/fs/resctrl
```

mount options are:

"cdp":
Enable code/data prioritization in L3 cache allocations.

"cdpl2":
Enable code/data prioritization in L2 cache allocations.

"mba_MBps":
Enable the MBA Software Controller(mba_sc) to specify MBA bandwidth in MBps

L2 and L3 CDP are controlled separately.

RDT features are orthogonal. A particular system may support only monitoring, only control, or both monitoring and control. Cache pseudo-locking is a unique way of using cache control to "pin" or "lock" data in the cache. Details can be found in "Cache Pseudo-Locking".

The mount succeeds if either of allocation or monitoring is present, but only those files and directories supported by the system will be created. For more details on the behavior of the interface during monitoring and allocation, see the "Resource alloc and monitor groups" section.

Info directory

The 'info' directory contains information about the enabled resources. Each resource has its own subdirectory. The subdirectory names reflect the resource names.

Each subdirectory contains the following files with respect to allocation:

Cache resource(L3/L2) subdirectory contains the following files related to allocation:

"num_closids":

The number of CLOSIDs which are valid for this resource. The kernel uses the smallest number of CLOSIDs of all enabled resources as limit.

"cbm_mask":

The bitmask which is valid for this resource. This mask is equivalent to 100%.

"min_cbm_bits":

The minimum number of consecutive bits which must be set when writing a mask.

"shareable_bits":

Bitmask of shareable resource with other executing entities (e.g. I/O). User can use this when setting up exclusive cache partitions. Note that some platforms support devices that have their own settings for cache use which can over-ride these bits.

"bit_usage":

Annotated capacity bitmasks showing how all instances of the resource are used. The legend is:

"0":
Corresponding region is unused. When the system's resources have been allocated and a "0" is found in "bit_usage" it is a sign that resources are wasted.
"H":
Corresponding region is used by hardware only but available for software use. If a resource has bits set in "shareable_bits" but not all of these bits appear in the resource groups' schematas then the bits appearing in "shareable_bits" but no resource group will be marked as "H".
"X":
Corresponding region is available for sharing and used by hardware and software. These are the bits that appear in "shareable_bits" as well as a resource group's allocation.
"S":
Corresponding region is used by software and available for sharing.
"E":
Corresponding region is used exclusively by one resource group. No sharing allowed.
"P":
Corresponding region is pseudo-locked. No sharing allowed.

Memory bandwidth(MB) subdirectory contains the following files with respect to allocation:

"min_bandwidth":

The minimum memory bandwidth percentage which user can request.

"bandwidth_gran":

The granularity in which the memory bandwidth percentage is allocated. The allocated b/w percentage is rounded off to the next control step available on the hardware. The available bandwidth control steps are: min_bandwidth + N * bandwidth_gran.

"delay_linear":

Indicates if the delay scale is linear or non-linear. This field is purely informational only.

"thread_throttle_mode":

Indicator on Intel systems of how tasks running on threads of a physical core are throttled in cases where they request different memory bandwidth percentages:

"max":
the smallest percentage is applied to all threads
"per-thread":
bandwidth percentages are directly applied to the threads running on the core

If RDT monitoring is available there will be an "L3_MON" directory with the following files:

"num_rmids":
The number of RMIDs available. This is the upper bound for how many "CTRL_MON" + "MON" groups can be created.
"mon_features":
Lists the monitoring events if monitoring is enabled for the resource.
"max_threshold_occupancy":
Read/write file provides the largest value (in bytes) at which a previously used LLC_occupancy counter can be considered for re-use.

Finally, in the top level of the "info" directory there is a file named "last_cmd_status". This is reset with every "command" issued via the file system (making new directories or writing to any of the control files). If the command was successful, it will read as "ok". If the command failed, it will provide more information that can be conveyed in the error returns from file operations. E.g.

```
# echo L3:0=f7 > schemata
bash: echo: write error: Invalid argument
# cat info/last_cmd_status
mask f7 has non-consecutive 1-bits
```

Resource alloc and monitor groups

Resource groups are represented as directories in the resctrl file system. The default group is the root directory which, immediately after mounting, owns all the tasks and cpus in the system and can make full use of all resources.

On a system with RDT control features additional directories can be created in the root directory that specify different amounts of each resource (see "schemata" below). The root and these additional top level directories are referred to as "CTRL_MON" groups below.

On a system with RDT monitoring the root directory and other top level directories contain a directory named "mon_groups" in which additional directories can be created to monitor subsets of tasks in the CTRL_MON group that is their ancestor. These are called "MON" groups in the rest of this document.

Removing a directory will move all tasks and cpus owned by the group it represents to the parent. Removing one of the created CTRL_MON groups will automatically remove all MON groups below it.

All groups contain the following files:

"tasks":
: Reading this file shows the list of all tasks that belong to this group. Writing a task id to the file will add a task to the group. If the group is a CTRL_MON group the task is removed from whichever previous CTRL_MON group owned the task and also from any MON group that owned the task. If the group is a MON group, then the task must already belong to the CTRL_MON parent of this group. The task is removed from any previous MON group.

"cpus":
: Reading this file shows a bitmask of the logical CPUs owned by this group. Writing a mask to this file will add and remove CPUs to/from this group. As with the tasks file a hierarchy is maintained where MON groups may only include CPUs owned by the parent CTRL_MON group. When the resource group is in pseudo-locked mode this file will only be readable, reflecting the CPUs associated with the pseudo-locked region.

"cpus_list":
: Just like "cpus", only using ranges of CPUs instead of bitmasks.

When control is enabled all CTRL_MON groups will also contain:

"schemata":
: A list of all the resources available to this group. Each resource has its own line and format - see below for details.

"size":
: Mirrors the display of the "schemata" file to display the size in bytes of each allocation instead of the bits representing the allocation.

"mode":
: The "mode" of the resource group dictates the sharing of its allocations. A "shareable" resource group allows sharing of its allocations while an "exclusive" resource group does not. A cache pseudo-locked region is created by first writing "pseudo-locksetup" to the "mode" file before writing the cache pseudo-locked region's schemata to the resource group's "schemata" file. On successful pseudo-locked region creation the mode will automatically change to "pseudo-locked".

When monitoring is enabled all MON groups will also contain:

"mon_data":
: This contains a set of files organized by L3 domain and by RDT event. E.g. on a system with two L3 domains there will be subdirectories "mon_L3_00" and "mon_L3_01". Each of these directories have one file per event (e.g. "llc_occupancy", "mbm_total_bytes", and "mbm_local_bytes"). In a MON group these files provide a read out of the current value of the event for all tasks in the group. In CTRL_MON groups these files provide the sum for all tasks in the CTRL_MON group and all tasks in MON groups. Please see example section for more details on usage.

Resource allocation rules

When a task is running the following rules define which resources are available to it:

1. If the task is a member of a non-default group, then the schemata for that group is used.
2. Else if the task belongs to the default group, but is running on a CPU that is assigned to some specific group, then the schemata for the CPU's group is used.
3. Otherwise the schemata for the default group is used.

Resource monitoring rules

1. If a task is a member of a MON group, or non-default CTRL_MON group then RDT events for the task will be reported in that group.
2. If a task is a member of the default CTRL_MON group, but is running on a CPU that is assigned to some specific group, then the RDT events for the task will be reported in that group.
3. Otherwise RDT events for the task will be reported in the root level "mon_data" group.

Notes on cache occupancy monitoring and control

When moving a task from one group to another you should remember that this only affects *new* cache allocations by the task. E.g. you may have a task in a monitor group showing 3 MB of cache occupancy. If you move to a new group and immediately check the occupancy of the old and new groups you will likely see that the old group is still showing 3 MB and the new group zero. When the task accesses locations still in cache from before the move, the h/w does not update any counters. On a busy system you will likely see the occupancy in the old group go down as cache lines are evicted and re-used while the occupancy in the new group rises as the task accesses memory and loads into the cache are counted based on membership in the new group.

The same applies to cache allocation control. Moving a task to a group with a smaller cache partition will not evict any cache lines. The process may continue to use them from the old partition.

Hardware uses CLOSid(Class of service ID) and an RMID(Resource monitoring ID) to identify a control group and a monitoring group respectively. Each of the resource groups are mapped to these IDs based on the kind of group. The number of CLOSid and

RMID are limited by the hardware and hence the creation of a "CTRL_MON" directory may fail if we run out of either CLOSID or RMID and creation of "MON" group may fail if we run out of RMIDs.

max_threshold_occupancy - generic concepts

Note that an RMID once freed may not be immediately available for use as the RMID is still tagged the cache lines of the previous user of RMID. Hence such RMIDs are placed on limbo list and checked back if the cache occupancy has gone down. If there is a time when system has a lot of limbo RMIDs but which are not ready to be used, user may see an -EBUSY during mkdir.

max_threshold_occupancy is a user configurable value to determine the occupancy at which an RMID can be freed.

Schemata files - general concepts

Each line in the file describes one resource. The line starts with the name of the resource, followed by specific values to be applied in each of the instances of that resource on the system.

Cache IDs

On current generation systems there is one L3 cache per socket and L2 caches are generally just shared by the hyperthreads on a core, but this isn't an architectural requirement. We could have multiple separate L3 caches on a socket, multiple cores could share an L2 cache. So instead of using "socket" or "core" to define the set of logical cpus sharing a resource we use a "Cache ID". At a given cache level this will be a unique number across the whole system (but it isn't guaranteed to be a contiguous sequence, there may be gaps). To find the ID for each logical CPU look in /sys/devices/system/cpu/cpu*/cache/index*/id

Cache Bit Masks (CBM)

For cache resources we describe the portion of the cache that is available for allocation using a bitmask. The maximum value of the mask is defined by each cpu model (and may be different for different cache levels). It is found using CPUID, but is also provided in the "info" directory of the resctrl file system in "info/{resource}/cbm_mask". Intel hardware requires that these masks have all the '1' bits in a contiguous block. So 0x3, 0x6 and 0xC are legal 4-bit masks with two bits set, but 0x5, 0x9 and 0xA are not. On a system with a 20-bit mask each bit represents 5% of the capacity of the cache. You could partition the cache into four equal parts with masks: 0x1f, 0x3e0, 0x7c00, 0xf8000.

Memory bandwidth Allocation and monitoring

For Memory bandwidth resource, by default the user controls the resource by indicating the percentage of total memory bandwidth.

The minimum bandwidth percentage value for each cpu model is predefined and can be looked up through "info/MB/min_bandwidth". The bandwidth granularity that is allocated is also dependent on the cpu model and can be looked up at "info/MB/bandwidth_gran". The available bandwidth control steps are: min_bw + N * bw_gran. Intermediate values are rounded to the next control step available on the hardware.

The bandwidth throttling is a core specific mechanism on some of Intel SKUs. Using a high bandwidth and a low bandwidth setting on two threads sharing a core may result in both threads being throttled to use the low bandwidth (see "thread_throttle_mode").

The fact that Memory bandwidth allocation(MBA) may be a core specific mechanism where as memory bandwidth monitoring(MBM) is done at the package level may lead to confusion when users try to apply control via the MBA and then monitor the bandwidth to see if the controls are effective. Below are such scenarios:

1. User may *not* see increase in actual bandwidth when percentage values are increased:

This can occur when aggregate L2 external bandwidth is more than L3 external bandwidth. Consider an SKL SKU with 24 cores on a package and where L2 external is 10GBps (hence aggregate L2 external bandwidth is 240GBps) and L3 external bandwidth is 100GBps. Now a workload with '20 threads, having 50% bandwidth, each consuming 5GBps' consumes the max L3 bandwidth of 100GBps although the percentage value specified is only 50% << 100%. Hence increasing the bandwidth percentage will not yield any more bandwidth. This is because although the L2 external bandwidth still has capacity, the L3 external bandwidth is fully used. Also note that this would be dependent on number of cores the benchmark is run on.

2. Same bandwidth percentage may mean different actual bandwidth depending on # of threads:

For the same SKU in #1, a 'single thread, with 10% bandwidth' and '4 thread, with 10% bandwidth' can consume upto 10GBps and 40GBps although they have same percentage bandwidth of 10%. This is simply because as threads start using more cores in an rdtgroup, the actual bandwidth may increase or vary although user specified bandwidth percentage is same.

In order to mitigate this and make the interface more user friendly, resctrl added support for specifying the bandwidth in MBps as well. The kernel underneath would use a software feedback mechanism or a "Software Controller(mba_sc)" which reads the actual bandwidth using MBM counters and adjust the memory bandwidth percentages to ensure:

```
"actual bandwidth < user specified bandwidth".
```

By default, the schemata would take the bandwidth percentage values where as user can switch to the "MBA software controller" mode using a mount option 'mba_MBps'. The schemata format is specified in the below sections.

L3 schemata file details (code and data prioritization disabled)

With CDP disabled the L3 schemata format is:

```
L3:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
```

L3 schemata file details (CDP enabled via mount option to resctrl)

When CDP is enabled L3 control is split into two separate resources so you can specify independent masks for code and data like this:

```
L3DATA:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
L3CODE:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
```

L2 schemata file details

CDP is supported at L2 using the 'cdpl2' mount option. The schemata format is either:

```
L2:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
```

or

L2DATA:<cache_id0>=<cbm>;<cache_id1>=<cbm>;... L2CODE:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...

Memory bandwidth Allocation (default mode)

Memory b/w domain is L3 cache.

```
MB:<cache_id0>=bandwidth0;<cache_id1>=bandwidth1;...
```

Memory bandwidth Allocation specified in MBps

Memory bandwidth domain is L3 cache.

```
MB:<cache_id0>=bw_MBps0;<cache_id1>=bw_MBps1;...
```

Reading/writing the schemata file

Reading the schemata file will show the state of all resources on all domains. When writing you only need to specify those values which you wish to change. E.g.

```
# cat schemata
L3DATA:0=fffff;1=fffff;2=fffff;3=fffff
L3CODE:0=fffff;1=fffff;2=fffff;3=fffff
# echo "L3DATA:2=3c0;" > schemata
# cat schemata
L3DATA:0=fffff;1=fffff;2=3c0;3=fffff
L3CODE:0=fffff;1=fffff;2=fffff;3=fffff
```

Cache Pseudo-Locking

CAT enables a user to specify the amount of cache space that an application can fill. Cache pseudo-locking builds on the fact that a CPU can still read and write data pre-allocated outside its current allocated area on a cache hit. With cache pseudo-locking, data can be preloaded into a reserved portion of cache that no application can fill, and from that point on will only serve cache hits. The cache pseudo-locked memory is made accessible to user space where an application can map it into its virtual address space and thus have a region of memory with reduced average read latency.

The creation of a cache pseudo-locked region is triggered by a request from the user to do so that is accompanied by a schemata of the region to be pseudo-locked. The cache pseudo-locked region is created as follows:

- Create a CAT allocation CLOSNEW with a CBM matching the schemata from the user of the cache region that will contain the pseudo-locked memory. This region must not overlap with any current CAT allocation/CLOS on the system and no future overlap with this cache region is allowed while the pseudo-locked region exists.
- Create a contiguous region of memory of the same size as the cache region.
- Flush the cache, disable hardware prefetchers, disable preemption.
- Make CLOSNEW the active CLOS and touch the allocated memory to load it into the cache.
- Set the previous CLOS as active.
- At this point the closid CLOSNEW can be released - the cache pseudo-locked region is protected as long as its CBM does not appear in any CAT allocation. Even though the cache pseudo-locked region will from this point on not appear in any CBM of any CLOS an application running with any CLOS will be able to access the memory in the pseudo-locked region since the region continues to serve cache hits.
- The contiguous region of memory loaded into the cache is exposed to user-space as a character device.

Cache pseudo-locking increases the probability that data will remain in the cache via carefully configuring the CAT feature and controlling application behavior. There is no guarantee that data is placed in cache. Instructions like INVD, WBINVD, CLFLUSH,

etc. can still evict “locked” data from cache. Power management C-states may shrink or power off cache. Deeper C-states will automatically be restricted on pseudo-locked region creation.

It is required that an application using a pseudo-locked region runs with affinity to the cores (or a subset of the cores) associated with the cache on which the pseudo-locked region resides. A sanity check within the code will not allow an application to map pseudo-locked memory unless it runs with affinity to cores associated with the cache on which the pseudo-locked region resides. The sanity check is only done during the initial mmap() handling, there is no enforcement afterwards and the application self needs to ensure it remains affine to the correct cores.

Pseudo-locking is accomplished in two stages:

1. During the first stage the system administrator allocates a portion of cache that should be dedicated to pseudo-locking. At this time an equivalent portion of memory is allocated, loaded into allocated cache portion, and exposed as a character device.
2. During the second stage a user-space application maps (mmap()) the pseudo-locked memory into its address space.

Cache Pseudo-Locking Interface

A pseudo-locked region is created using the resctrl interface as follows:

1. Create a new resource group by creating a new directory in /sys/fs/resctrl.
2. Change the new resource group's mode to "pseudo-locksetup" by writing "pseudo-locksetup" to the "mode" file.
3. Write the schemata of the pseudo-locked region to the "schemata" file. All bits within the schemata should be "unused" according to the "bit_usage" file.

On successful pseudo-locked region creation the "mode" file will contain "pseudo-locked" and a new character device with the same name as the resource group will exist in /dev/pseudo_lock. This character device can be mmap()'ed by user space in order to obtain access to the pseudo-locked memory region.

An example of cache pseudo-locked region creation and usage can be found below.

Cache Pseudo-Locking Debugging Interface

The pseudo-locking debugging interface is enabled by default (if CONFIG_DEBUG_FS is enabled) and can be found in /sys/kernel/debug/resctrl.

There is no explicit way for the kernel to test if a provided memory location is present in the cache. The pseudo-locking debugging interface uses the tracing infrastructure to provide two ways to measure cache residency of the pseudo-locked region:

1. Memory access latency using the pseudo_lock_mem_latency tracepoint. Data from these measurements are best visualized using a hist trigger (see example below). In this test the pseudo-locked region is traversed at a stride of 32 bytes while hardware prefetchers and preemption are disabled. This also provides a substitute visualization of cache hits and misses.
2. Cache hit and miss measurements using model specific precision counters if available. Depending on the levels of cache on the system the pseudo_lock_l2 and pseudo_lock_l3 tracepoints are available.

When a pseudo-locked region is created a new debugfs directory is created for it in debugfs as /sys/kernel/debug/resctrl/<newdir>. A single write-only file, pseudo_lock_measure, is present in this directory. The measurement of the pseudo-locked region depends on the number written to this debugfs file:

1:
: writing "1" to the pseudo_lock_measure file will trigger the latency measurement captured in the pseudo_lock_mem_latency tracepoint. See example below.

2:
: writing "2" to the pseudo_lock_measure file will trigger the L2 cache residency (cache hits and misses) measurement captured in the pseudo_lock_l2 tracepoint. See example below.

3:
: writing "3" to the pseudo_lock_measure file will trigger the L3 cache residency (cache hits and misses) measurement captured in the pseudo_lock_l3 tracepoint.

All measurements are recorded with the tracing infrastructure. This requires the relevant tracepoints to be enabled before the measurement is triggered.

Example of latency debugging interface

In this example a pseudo-locked region named "newlock" was created. Here is how we can measure the latency in cycles of reading from this region and visualize this data with a histogram that is available if CONFIG_HIST_TRIGGERS is set:

```
# :> /sys/kernel/debug/tracing/trace
# echo 'hist:keys=latency' > /sys/kernel/debug/tracing/events/resctrl/pseudo_lock_mem_latency/trigger
# echo 1 > /sys/kernel/debug/tracing/events/resctrl/pseudo_lock_mem_latency/enable
# echo 1 > /sys/kernel/debug/resctrl/newlock/pseudo_lock_measure
# echo 0 > /sys/kernel/debug/tracing/events/resctrl/pseudo_lock_mem_latency/enable
# cat /sys/kernel/debug/tracing/events/resctrl/pseudo_lock_mem_latency/hist

# event histogram
```

```
#
# trigger info: hist:keys=latency:vals=hitcount:sort=hitcount:size=2048 [active]
#

{ latency:        456 } hitcount:          1
{ latency:         50 } hitcount:         83
{ latency:         36 } hitcount:         96
{ latency:         44 } hitcount:        174
{ latency:         48 } hitcount:        195
{ latency:         46 } hitcount:        262
{ latency:         42 } hitcount:        693
{ latency:         40 } hitcount:       3204
{ latency:         38 } hitcount:       3484

Totals:
    Hits: 8192
    Entries: 9
  Dropped: 0
```

Example of cache hits/misses debugging

In this example a pseudo-locked region named "newlock" was created on the L2 cache of a platform. Here is how we can obtain details of the cache hits and misses using the platform's precision counters.

```
# :> /sys/kernel/debug/tracing/trace
# echo 1 > /sys/kernel/debug/tracing/events/resctrl/pseudo_lock_l2/enable
# echo 2 > /sys/kernel/debug/resctrl/newlock/pseudo_lock_measure
# echo 0 > /sys/kernel/debug/tracing/events/resctrl/pseudo_lock_l2/enable
# cat /sys/kernel/debug/tracing/trace

# tracer: nop
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
pseudo_lock_mea-1672  [002] ....  3132.860500: pseudo_lock_l2: hits=4097 miss=0
```

Examples for RDT allocation usage

1. Example 1

On a two socket machine (one L3 cache per socket) with just four bits for cache bit masks, minimum b/w of 10% with a memory bandwidth granularity of 10%.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir p0 p1
# echo "L3:0=3;1=c\nMB:0=50;1=50" > /sys/fs/resctrl/p0/schemata
# echo "L3:0=3;1=3\nMB:0=50;1=50" > /sys/fs/resctrl/p1/schemata
```

The default resource group is unmodified, so we have access to all parts of all caches (its schemata file reads "L3:0=f;1=f").

Tasks that are under the control of group "p0" may only allocate from the "lower" 50% on cache ID 0, and the "upper" 50% of cache ID 1. Tasks in group "p1" use the "lower" 50% of cache on both sockets.

Similarly, tasks that are under the control of group "p0" may use a maximum memory b/w of 50% on socket0 and 50% on socket 1. Tasks in group "p1" may also use 50% memory b/w on both sockets. Note that unlike cache masks, memory b/w cannot specify whether these allocations can overlap or not. The allocations specifies the maximum b/w that the group may be able to use and the system admin can configure the b/w accordingly.

If resctrl is using the software controller (mba_sc) then user can enter the max b/w in MB rather than the percentage values.

```
# echo "L3:0=3;1=c\nMB:0=1024;1=500" > /sys/fs/resctrl/p0/schemata
# echo "L3:0=3;1=3\nMB:0=1024;1=500" > /sys/fs/resctrl/p1/schemata
```

In the above example the tasks in "p1" and "p0" on socket 0 would use a max b/w of 1024MB where as on socket 1 they would use 500MB.

2. Example 2

Again two sockets, but this time with a more realistic 20-bit mask.

Two real time tasks pid=1234 running on processor 0 and pid=5678 running on processor 1 on socket 0 on a 2-socket and dual core machine. To avoid noisy neighbors, each of the two real-time tasks exclusively occupies one quarter of L3 cache on socket 0.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
```

First we reset the schemata for the default group so that the "upper" 50% of the L3 cache on socket 0 and 50% of memory b/w cannot be used by ordinary tasks:

```
# echo "L3:0=3ff;1=fffff\nMB:0=50;1=100" > schemata
```

Next we make a resource group for our first real time task and give it access to the "top" 25% of the cache on socket 0.

```
# mkdir p0
# echo "L3:0=f8000;1=fffff" > p0/schemata
```

Finally we move our first real time task into this resource group. We also use taskset(1) to ensure the task always runs on a dedicated CPU on socket 0. Most uses of resource groups will also constrain which processors tasks run on.

```
# echo 1234 > p0/tasks
# taskset -cp 1 1234
```

Ditto for the second real time task (with the remaining 25% of cache):

```
# mkdir p1
# echo "L3:0=7c00;1=fffff" > p1/schemata
# echo 5678 > p1/tasks
# taskset -cp 2 5678
```

For the same 2 socket system with memory b/w resource and CAT L3 the schemata would look like(Assume min_bandwidth 10 and bandwidth_gran is 10):

For our first real time task this would request 20% memory b/w on socket 0.

```
# echo -e "L3:0=f8000;1=fffff\nMB:0=20;1=100" > p0/schemata
```

For our second real time task this would request an other 20% memory b/w on socket 0.

```
# echo -e "L3:0=f8000;1=fffff\nMB:0=20;1=100" > p0/schemata
```

3. Example 3

A single socket system which has real-time tasks running on core 4-7 and non real-time workload assigned to core 0-3. The real-time tasks share text and data, so a per task association is not required and due to interaction with the kernel it's desired that the kernel on these cores shares L3 with the tasks.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
```

First we reset the schemata for the default group so that the "upper" 50% of the L3 cache on socket 0, and 50% of memory bandwidth on socket 0 cannot be used by ordinary tasks:

```
# echo "L3:0=3ff\nMB:0=50" > schemata
```

Next we make a resource group for our real time cores and give it access to the "top" 50% of the cache on socket 0 and 50% of memory bandwidth on socket 0.

```
# mkdir p0
# echo "L3:0=ffc00\nMB:0=50" > p0/schemata
```

Finally we move core 4-7 over to the new group and make sure that the kernel and the tasks running there get 50% of the cache. They should also get 50% of memory bandwidth assuming that the cores 4-7 are SMT siblings and only the real time threads are scheduled on the cores 4-7.

```
# echo F0 > p0/cpus
```

4. Example 4

The resource groups in previous examples were all in the default "shareable" mode allowing sharing of their cache allocations. If one resource group configures a cache allocation then nothing prevents another resource group to overlap with that allocation.

In this example a new exclusive resource group will be created on a L2 CAT system with two L2 cache instances that can be configured with an 8-bit capacity bitmask. The new exclusive resource group will be configured to use 25% of each cache instance.

```
# mount -t resctrl resctrl /sys/fs/resctrl/
# cd /sys/fs/resctrl
```

First, we observe that the default group is configured to allocate to all L2 cache:

```
# cat schemata
L2:0=ff;1=ff
```

We could attempt to create the new resource group at this point, but it will fail because of the overlap with the schemata of the default group:

```
# mkdir p0
# echo 'L2:0=0x3;1=0x3' > p0/schemata
```

```
# cat p0/mode
shareable
# echo exclusive > p0/mode
-sh: echo: write error: Invalid argument
# cat info/last_cmd_status
schemata overlaps
```

To ensure that there is no overlap with another resource group the default resource group's schemata has to change, making it possible for the new resource group to become exclusive.

```
# echo 'L2:0=0xfc;1=0xfc' > schemata
# echo exclusive > p0/mode
# grep . p0/*
p0/cpus:0
p0/mode:exclusive
p0/schemata:L2:0=03;1=03
p0/size:L2:0=262144;1=262144
```

A new resource group will on creation not overlap with an exclusive resource group:

```
# mkdir p1
# grep . p1/*
p1/cpus:0
p1/mode:shareable
p1/schemata:L2:0=fc;1=fc
p1/size:L2:0=786432;1=786432
```

The bit_usage will reflect how the cache is used:

```
# cat info/L2/bit_usage
0=SSSSSSEE;1=SSSSSSEE
```

A resource group cannot be forced to overlap with an exclusive resource group:

```
# echo 'L2:0=0x1;1=0x1' > p1/schemata
-sh: echo: write error: Invalid argument
# cat info/last_cmd_status
overlaps with exclusive group
```

Example of Cache Pseudo-Locking

Lock portion of L2 cache from cache id 1 using CBM 0x3. Pseudo-locked region is exposed at /dev/pseudo_lock/newlock that can be provided to application for argument to mmap().

```
# mount -t resctrl resctrl /sys/fs/resctrl/
# cd /sys/fs/resctrl
```

Ensure that there are bits available that can be pseudo-locked, since only unused bits can be pseudo-locked the bits to be pseudo-locked needs to be removed from the default resource group's schemata:

```
# cat info/L2/bit_usage
0=SSSSSSSS;1=SSSSSSSS
# echo 'L2:1=0xfc' > schemata
# cat info/L2/bit_usage
0=SSSSSSSS;1=SSSSSS00
```

Create a new resource group that will be associated with the pseudo-locked region, indicate that it will be used for a pseudo-locked region, and configure the requested pseudo-locked region capacity bitmask:

```
# mkdir newlock
# echo pseudo-locksetup > newlock/mode
# echo 'L2:1=0x3' > newlock/schemata
```

On success the resource group's mode will change to pseudo-locked, the bit_usage will reflect the pseudo-locked region, and the character device exposing the pseudo-locked region will exist:

```
# cat newlock/mode
pseudo-locked
# cat info/L2/bit_usage
0=SSSSSSSS;1=SSSSSSPP
# ls -l /dev/pseudo_lock/newlock
crw------- 1 root root 243, 0 Apr  3 05:01 /dev/pseudo_lock/newlock
```

```
/*
* Example code to access one page of pseudo-locked cache region
* from user space.
*/
#define _GNU_SOURCE
#include <fcntl.h>
#include <sched.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

/*
* It is required that the application runs with affinity to only
* cores associated with the pseudo-locked region. Here the cpu
* is hardcoded for convenience of example.
*/
static int cpuid = 2;

int main(int argc, char *argv[])
{
  cpu_set_t cpuset;
  long page_size;
  void *mapping;
  int dev_fd;
  int ret;

  page_size = sysconf(_SC_PAGESIZE);

  CPU_ZERO(&cpuset);
  CPU_SET(cpuid, &cpuset);
  ret = sched_setaffinity(0, sizeof(cpuset), &cpuset);
  if (ret < 0) {
    perror("sched_setaffinity");
    exit(EXIT_FAILURE);
  }

  dev_fd = open("/dev/pseudo_lock/newlock", O_RDWR);
  if (dev_fd < 0) {
    perror("open");
    exit(EXIT_FAILURE);
  }

  mapping = mmap(0, page_size, PROT_READ | PROT_WRITE, MAP_SHARED,
          dev_fd, 0);
  if (mapping == MAP_FAILED) {
    perror("mmap");
    close(dev_fd);
    exit(EXIT_FAILURE);
  }

  /* Application interacts with pseudo-locked memory @mapping */

  ret = munmap(mapping, page_size);
  if (ret < 0) {
    perror("munmap");
    close(dev_fd);
    exit(EXIT_FAILURE);
  }

  close(dev_fd);
  exit(EXIT_SUCCESS);
}
```

Locking between applications

Certain operations on the resctrl filesystem, composed of read/writes to/from multiple files, must be atomic.

As an example, the allocation of an exclusive reservation of L3 cache involves:

1. Read the cbmmasks from each directory or the per-resource "bit_usage"
2. Find a contiguous set of bits in the global CBM bitmask that is clear in any of the directory cbmmasks
3. Create a new directory
4. Set the bits found in step 2 to the new directory "schemata" file

If two applications attempt to allocate space concurrently then they can end up allocating the same bits so the reservations are shared instead of exclusive.

To coordinate atomic operations on the resctrlfs and to avoid the problem above, the following locking procedure is recommended:

Locking is based on flock, which is available in libc and also as a shell script command

Write lock:

A. Take flock(LOCK_EX) on /sys/fs/resctrl
B. Read/write the directory structure.
C. funlock

Read lock:

A. Take flock(LOCK_SH) on /sys/fs/resctrl
B. If success read the directory structure.
C. funlock

Example with bash:

```
# Atomically read directory structure
$ flock -s /sys/fs/resctrl/ find /sys/fs/resctrl

# Read directory contents and create new subdirectory

$ cat create-dir.sh
find /sys/fs/resctrl/ > output.txt
mask = function-of(output.txt)
mkdir /sys/fs/resctrl/newres/
echo mask > /sys/fs/resctrl/newres/schemata

$ flock /sys/fs/resctrl/ ./create-dir.sh
```

Example with C:

```
/*
* Example code do take advisory locks
* before accessing resctrl filesystem
*/
#include <sys/file.h>
#include <stdlib.h>

void resctrl_take_shared_lock(int fd)
{
  int ret;

  /* take shared lock on resctrl filesystem */
  ret = flock(fd, LOCK_SH);
  if (ret) {
    perror("flock");
    exit(-1);
  }
}

void resctrl_take_exclusive_lock(int fd)
{
  int ret;

  /* release lock on resctrl filesystem */
  ret = flock(fd, LOCK_EX);
  if (ret) {
    perror("flock");
    exit(-1);
  }
}

void resctrl_release_lock(int fd)
{
  int ret;

  /* take shared lock on resctrl filesystem */
  ret = flock(fd, LOCK_UN);
  if (ret) {
    perror("flock");
    exit(-1);
  }
}

void main(void)
{
  int fd, ret;

  fd = open("/sys/fs/resctrl", O_DIRECTORY);
  if (fd == -1) {
    perror("open");
    exit(-1);
  }
  resctrl_take_shared_lock(fd);
  /* code to read directory contents */
  resctrl_release_lock(fd);

  resctrl_take_exclusive_lock(fd);
```

```
    /* code to read and write directory contents */
    resctrl_release_lock(fd);
}
```

Examples for RDT Monitoring along with allocation usage

Reading monitored data

Reading an event file (for ex: mon_data/mon_L3_00/llc_occupancy) would show the current snapshot of LLC occupancy of the corresponding MON group or CTRL_MON group.

Example 1 (Monitor CTRL_MON group and subset of tasks in CTRL_MON group)

On a two socket machine (one L3 cache per socket) with just four bits for cache bit masks:

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir p0 p1
# echo "L3:0=3;1=c" > /sys/fs/resctrl/p0/schemata
# echo "L3:0=3;1=3" > /sys/fs/resctrl/p1/schemata
# echo 5678 > p1/tasks
# echo 5679 > p1/tasks
```

The default resource group is unmodified, so we have access to all parts of all caches (its schemata file reads "L3:0=f;1=f").

Tasks that are under the control of group "p0" may only allocate from the "lower" 50% on cache ID 0, and the "upper" 50% of cache ID 1. Tasks in group "p1" use the "lower" 50% of cache on both sockets.

Create monitor groups and assign a subset of tasks to each monitor group.

```
# cd /sys/fs/resctrl/p1/mon_groups
# mkdir m11 m12
# echo 5678 > m11/tasks
# echo 5679 > m12/tasks
```

fetch data (data shown in bytes)

```
# cat m11/mon_data/mon_L3_00/llc_occupancy
16234000
# cat m11/mon_data/mon_L3_01/llc_occupancy
14789000
# cat m12/mon_data/mon_L3_00/llc_occupancy
16789000
```

The parent ctrl_mon group shows the aggregated data.

```
# cat /sys/fs/resctrl/p1/mon_data/mon_l3_00/llc_occupancy
31234000
```

Example 2 (Monitor a task from its creation)

On a two socket machine (one L3 cache per socket):

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir p0 p1
```

An RMID is allocated to the group once its created and hence the <cmd> below is monitored from its creation.

```
# echo $$ > /sys/fs/resctrl/p1/tasks
# <cmd>
```

Fetch the data:

```
# cat /sys/fs/resctrl/p1/mon_data/mon_l3_00/llc_occupancy
31789000
```

Example 3 (Monitor without CAT support or before creating CAT groups)

Assume a system like HSW has only CQM and no CAT support. In this case the resctrl will still mount but cannot create CTRL_MON directories. But user can create different MON groups within the root group thereby able to monitor all tasks including kernel threads.

This can also be used to profile jobs cache size footprint before being able to allocate them to different allocation groups.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir mon_groups/m01
# mkdir mon_groups/m02
```

```
# echo 3478 > /sys/fs/resctrl/mon_groups/m01/tasks
# echo 2467 > /sys/fs/resctrl/mon_groups/m02/tasks
```

Monitor the groups separately and also get per domain data. From the below its apparent that the tasks are mostly doing work on domain(socket) 0.

```
# cat /sys/fs/resctrl/mon_groups/m01/mon_L3_00/llc_occupancy
31234000
# cat /sys/fs/resctrl/mon_groups/m01/mon_L3_01/llc_occupancy
34555
# cat /sys/fs/resctrl/mon_groups/m02/mon_L3_00/llc_occupancy
31234000
# cat /sys/fs/resctrl/mon_groups/m02/mon_L3_01/llc_occupancy
32789
```

Example 4 (Monitor real time tasks)

A single socket system which has real time tasks running on cores 4-7 and non real time tasks on other cpus. We want to monitor the cache occupancy of the real time threads on these cores.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir p1
```

Move the cpus 4-7 over to p1:

```
# echo f0 > p1/cpus
```

View the llc occupancy snapshot:

```
# cat /sys/fs/resctrl/p1/mon_data/mon_L3_00/llc_occupancy
11234000
```

Intel RDT Errata

Intel MBM Counters May Report System Memory Bandwidth Incorrectly

Errata SKX99 for Skylake server and BDF102 for Broadwell server.

Problem: Intel Memory Bandwidth Monitoring (MBM) counters track metrics according to the assigned Resource Monitor ID (RMID) for that logical core. The IA32_QM_CTR register (MSR 0xC8E), used to report these metrics, may report incorrect system bandwidth for certain RMID values.

Implication: Due to the errata, system memory bandwidth may not match what is reported.

Workaround: MBM total and local readings are corrected according to the following correction factor table:

core count	rmid count	rmid threshold	correction factor
1	8	0	1.000000
2	16	0	1.000000
3	24	15	0.969650
4	32	0	1.000000
6	48	31	0.969650
7	56	47	1.142857
8	64	0	1.000000
9	72	63	1.185115
10	80	63	1.066553
11	88	79	1.454545
12	96	0	1.000000
13	104	95	1.230769
14	112	95	1.142857
15	120	95	1.066667
16	128	0	1.000000
17	136	127	1.254863
18	144	127	1.185255
19	152	0	1.000000
20	160	127	1.066667
21	168	0	1.000000
22	176	159	1.454334
23	184	0	1.000000
24	192	127	0.969744

25	200	191	1.280246
26	208	191	1.230921
27	216	0	1.000000
28	224	191	1.143118

If rmid > rmid threshold, MBM total and local values should be multiplied by the correction factor.

See:

1. Erratum SKX99 in Intel Xeon Processor Scalable Family Specification Update: http://web.archive.org/web/20200716124958/https://www.intel.com/content/www/us/en/processors/xeon/scalable/xeon-scalable-spec-update.html

2. Erratum BDF102 in Intel Xeon E5-2600 v4 Processor Product Family Specification Update: http://web.archive.org/web/20191125200531/https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v4-spec-update.pdf

3. The errata in Intel Resource Director Technology (Intel RDT) on 2nd Generation Intel Xeon Scalable Processors Reference Manual: https://software.intel.com/content/www/us/en/develop/articles/intel-resource-director-technology-rdt-reference-manual.html

for further information.