Variorum Documentation

Release 0.7.0

LLNS

Jan 17, 2024

BASICS

1 Variorum Project Resources **3**

2 Lead Developers **5**

3 Code Of Conduct **7**

4 Acknowledgments **9**

5 Variorum Documentation **11**
5.1 Quick Start Guide . . . 11
5.2 Building Variorum . . . 12
5.3 Variorum Overview . . . 15
5.4 Variorum API . . . 16
5.5 Example Programs . . . 19
5.6 Supported Platform Documentation . . . 22
5.7 Monitoring Binaries with Variorum . . . 51
5.8 Variorum Utilities . . . 52
5.9 Variorum Print Functions . . . 54
5.10 Variorum Cap Functions . . . 63
5.11 Variorum JSON-Support Functions . . . 66
5.12 Variorum Enable/Disable Functions . . . 70
5.13 Variorum Topology Functions . . . 71
5.14 JSON API . . . 71
5.15 Contributing Guide . . . 72
5.16 Variorum Developer Documentation . . . 75
5.17 Variorum Unit Tests . . . 76
5.18 Integrating with Variorum . . . 77
5.19 ECP Argo Project . . . 78
5.20 HPC PowerStack Initiative . . . 79
5.21 Publications and Presentations . . . 80
5.22 Contributors . . . 81
5.23 Contributor Covenant Code of Conduct . . . 82
5.24 Releases . . . 83
5.25 License Info . . . 84

6 Indices and tables **87**

Index **89**

Variorum is an extensible, vendor-neutral library for exposing power and performance capabilities of low-level hardware dials across diverse architectures in a user-friendly manner. It is part of the *ECP Argo Project*, and is a key component for node-level power management in the *HPC PowerStack Initiative*. Variorum provides vendor-neutral APIs such that the user can query or control hardware dials without needing to know the underlying vendor's implementation (for example, model-specific registers or sensor interfaces). These APIs enable application developers to gain a better understanding of power, energy, and performance through various metrics. Additionally, the APIs may enable system software to control hardware dials to optimize for a particular goal. Variorum focuses on ease of use and reduced integration burden in applications, which it accomplishes by providing:

- Examples which demonstrate how to use Variorum in a stand-alone program.
- A performance analysis sampler that runs alongside the application.
- A JSON API to allow integration with higher-level system software, such as job schedulers, distributed monitoring frameworks, or application-level runtime systems.

CHAPTER

ONE

VARIORUM PROJECT RESOURCES

Online Documentation https://variorum.readthedocs.io/

Github Source Repo http://github.com/llnl/variorum

Issue Tracker http://github.com/llnl/variorum/issues

CHAPTER

TWO

LEAD DEVELOPERS

- Stephanie Brink (LLNL)
- Aniruddha Marathe (LLNL)
- Tapasya Patki (LLNL)
- Barry Rountree (LLNL)
- Kathleen Shoga (LLNL)

CHAPTER

THREE

CODE OF CONDUCT

See *Contributor Covenant Code of Conduct*.

CHAPTER

FOUR

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

CHAPTER

FIVE

VARIORUM DOCUMENTATION

5.1 Quick Start Guide

The simplest way to install Variorum is using the default CMake settings. Building Variorum requires the `hwloc` and `jansson` libraries. The default build targets the Intel platform and assumes `msr-safe` kernel module access.

```
# install hwloc and jansson dependencies
sudo apt-get install libhwloc15 libhwloc-dev libjansson4 libjansson-dev

git clone https://github.com/LLNL/variorum.git

cd variorum
mkdir build install
cd build

cmake -DCMAKE_INSTALL_PREFIX=../install ../src
make -j8
make install
```

Note that `HWLOC_DIR` and `JANSSON_DIR` may need to be set correctly if installing from scratch. Variorum also supports `host-config` files, which make the build process easier by setting all necessary paths in one place. Details on using host configuration files can be found here.

Please ensure that the dependencies for each platform are met before building Variorum. These include the kernel module `msr-safe` for Intel systems, `msr`, `amd_energy_driver` and HSMP driver for AMD sytems, OPAL firmware and sensors for IBM, and NVML for NVIDIA. Details of each of these can be found in the respective vendor pages, see *Supported Platform Documentation*.

For more details about building and installing Variorum, see *Building Variorum*, which provides detailed information about building Variorum for specific hosts, Variorum's other CMake options and installing with `spack`.

Function-level descriptions of Variorum's APIs as well as the architectures that have implementations in Variorum are provided in the following sections:

- *Variorum Print Functions*
- *Variorum Cap Functions*
- *Variorum JSON-Support Functions*
- *Variorum Enable/Disable Functions*
- *Variorum Topology Functions*
- *JSON API*

For beginners, the ECP Variorum Lecture Series is beneficial.

5.2 Building Variorum

Variorum can be built from source with CMake or with `spack`. Building Variorum creates the `libvariorum` library, the `powmon` monitoring tool, and Variorum examples.

5.2.1 Build Dependencies

The build dependencies for a **minimal build** with no parallel components require the following:

- C
- hwloc
- jansson
- Access to vendor-specific drivers or modules (see *Supported Platform Documentation*).

For a **parallel build** with MPI/OpenMP, Variorum depends on:

- rankstr (only required for MPI/OpenMP builds)

The CMake variables (`ENABLE_MPI={ON,OFF}` and `ENABLE_OPENMP={ON,OFF}`) control the building of parallel examples. If `ENABLE_MPI=ON`, an MPI compiler is required.

hwloc (Required)

hwloc is an open-source project providing a portable abstraction of the hierarchical topology of modern architectures.

Variorum leverages hwloc for detecting hardware topology. When reading/writing a register on a particular hardware thread, hwloc can map that to the correct physical socket.

jansson (Required)

jansson is an open-source C library for encoding, decoding and manipulating JSON data.

Variorum leverages JANSSON to provide a JSON-based API that can retrieve power data for external tools/software.

rankstr (Optional)

rankstr is an open-source C library providing functions that identify unique strings across an MPI communicator.

Variorum leverages rankstr to split a communicator into subcommunicators by hostname. The allows for a single control or monitor process in Variorum to for example, enforce a power or frequency limit on a node or to print the hardware counters once on a node.

5.2.2 Building with CMake

Variorum can be built and installed as follows after cloning from GitHub:

```
# install hwloc and jansson dependencies
sudo apt-get install libhwloc15 libhwloc-dev libjansson4 libjansson-dev

git clone https://github.com/llnl/variorum

cd variorum
mkdir build install
cd build

cmake -DCMAKE_INSTALL_PREFIX=../install ../src
make -j8
make install
```

5.2.3 Host Config Files

To handle build options, third party library paths, etc., we rely on CMake's initial-cache file mechanism. We call these initial-cache files `host-config` files, as we typically create a file for each platform or specific hosts if necessary. These can be passed to CMake via the `-C` command line option as shown below:

```
cmake {other options} -C ../host-configs/{config_file}.cmake ../src
```

An example is provided in `host-configs/boilerplate.cmake` to create your own configuration file. Example configuration files named by machine hostname, the `SYS_TYPE` environment variable, and platform name (via `uname`) are also provided in the `host-configs` directory. These files use standard CMake commands. CMake `set` commands need to specify the root cache path as follows:

```
set(CMAKE_VARIABLE_NAME {VALUE} CACHE PATH "")
```

5.2.4 CMake Build Options

Variorum's build system supports the following CMake options:

- `HWLOC_DIR` - Path to an HWLOC install.
- `JANSSON_DIR` - Path to a JANSSON install.
- `SPHINX_EXECUTABLE` - Path to sphinx-build binary (required for documentation).
- `VARIORUM_WITH_AMD_CPU (default=OFF)` - Enable Variorum build for AMD CPU architecture.
- `VARIORUM_WITH_NVIDIA_GPU (default=OFF)` - Enable Variorum build for Nvidia GPU architecture.
- `VARIORUM_WITH_IBM_CPU (default=OFF)` - Enable Variorum build for IBM CPU architecture.
- `VARIORUM_WITH_ARM_CPU (default=OFF)` - Enable Variorum build for ARM CPU architecture.
- `VARIORUM_WITH_INTEL_CPU (default=ON)` - Enable Variorum build for Intel CPU architecture.
- `VARIORUM_WITH_INTEL_GPU (default=OFF)` - Enable Variorum build for Intel discrete GPU architecture.
- `ENABLE_FORTRAN (default=ON)` - Enable Fortran compiler for building example integration with Fortran application, Fortran compiler must exist.

- `ENABLE_PYTHON (default=ON)` - Enable Python wrappers for adding PyVariorum examples.
- `ENABLE_MPI (default=OFF)` - Enable MPI compiler for building MPI examples, MPI compiler must exist.
- `ENABLE_OPENMP (default=ON)` - Enable OpenMP extensions for building OpenMP examples.
- `ENABLE_WARNINGS (default=OFF)` - Build with compiler warning flags -Wall -Wextra -Werror, used primarily by developers.
- `BUILD_DOCS (default=ON)` - Controls if the Variorum documentation is built (when sphinx and doxygen are found).
- `BUILD_SHARED_LIBS (default=ON)` - Controls if shared (ON) or static (OFF) libraries are built.
- `BUILD_TESTS (default=ON)` - Controls if unit tests are built.
- `VARIORUM_DEBUG (default=OFF)` - Enable Variorum debug statements, useful if values are not translating correctly.
- `USE_MSR_SAFE_BEFORE_1_5_0 (default=OFF)` - Use msr-safe prior to v1.5.0, dependency of Intel architectures for accessing counters from userspace.

5.2.5 Building on Multiple Architectures

Several HPC systems have compute nodes that are composed of components or hardware from different vendors. An example is the Sierra supercomputer at LLNL, where the CPUs are based on IBM platform (Power9), and the GPUs are from NVIDIA (Volta). Because power and energy dials are vendor-specific, we need to enable a *multi-architecture* build for such platforms. As of version 0.7, we have added support for such builds within Variorum. These can now be specified with turning the CMake options for the associated devices, for example. `VARIORUM_WITH_NVIDIA_GPU` and `VARIORUM_WITH_IBM_CPU` can both be set to `ON` during the same build.

Support for multi-architecture builds is still new and limited, especially with the JSON-based APIs. We are working toward adding more features in the near future.

5.2.6 Building with Spack

To install Variorum with all options (and also build all of its dependencies as necessary) run:

```
spack install variorum
```

The Variorum spack package provides several variants that customize the options and dependencies used to build Variorum (see table below). Variants are enabled using + and disabled using ~.

Variant	Description	Default
shared	Build Variorum as shared library	ON (+shared)
docs	Build Variorum's Documentation	OFF (~docs)
log	Enable Variorum's logging infrastructure	OFF (~log)
build_type	Specify build type	Release with Debug Info (build_type=RelWithDebugInfo)

5.2.7 Debugging

Setting the `VARIORUM_LOG` environment variable at runtime to `VARIORUM_LOG=1` will print out debugging information.

5.3 Variorum Overview

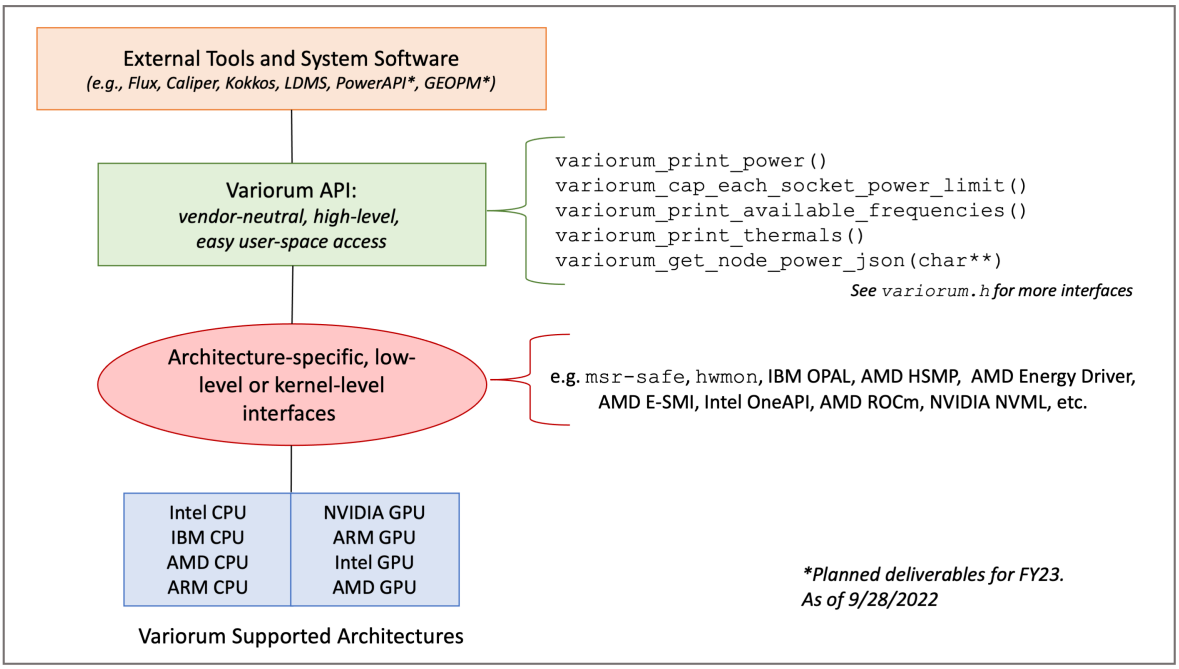
Variorum is a production-grade software infrastructure for exposing low-level control and monitoring of a system's underlying hardware features, with a focus on power, energy and thermal dials. It is a significant evolution based on an older open-source library developed at LLNL, called `libmsr`. Variorum is vendor-neutral, and can easily be ported to different hardware devices, as well as different generations within a particular device. More specifically, Variorum's flexible design supports a set of features that may exist on one generation of hardware, but not on another.

5.3.1 Design Principles

To guide the development of Variorum, we focused on a set of important requirements extracted from our learnings with the development of `libmsr`. Here are Variorum's requirements:

- **Create device-agnostic APIs**: We do not want to require the user to have to know or understand how to interface with each device. The library is built for a target architecture, which may be heterogeneous, and can collect data from each device through a single front-facing interface.
- **Provide a simple interface**: We want users and tool developers to not only collect information from the underlying hardware, but also to have the ability to control various features.
- **Ease in extending to new devices and generations within a device**: Variorum makes it easy to support new features, deprecate old features among generations of devices, and adapt features that may have different domains of control from one generation to another (i.e., sockets, cores, threads).

5.3.2 System Diagram



5.4 Variorum API

Variorum supports vendor-neutral power and energy management through its rich API. Please refer to the top-level API, as well as the specific descriptions of the JSON API and the Best Effort Power Capping API. The JSON API allows system software interacting with Variorum to obtain data in a portable, vendor-neutral manner.

5.4.1 Top-level API

The top-level API for Variorum is in the `variorum.h` header file. Function-level descriptions as well as the architectures that have implementations in Variorum are described in the following sections:

- *Variorum Print Functions*
- *Variorum Cap Functions*
- *Variorum JSON-Support Functions*
- *Variorum Enable/Disable Functions*
- *Variorum Topology Functions*
- *JSON API*

5.4.2 Variorum Wrappers

As of v0.6.0, Variorum also supports Fortran and Python APIs for Variorum, these can be found in the `src/wrappers` directory. By default, these wrappers will be enabled. The Fortran wrapper is built and installed if Fortran is found and enabled. For the Python module (called `pyVariorum`), a `pip` based install or setting of `PYTHONPATH` is needed. Please refer to the `README` in the `src/wrappers/python` directory for the details. Examples on the usage of these wrappers can be found in the `src/examples/fortran-examples` and the `src/examples/python-examples` directories, respectively.

5.4.3 JSON API

The current JSON API depends on the JANSSON-C library and has a vendor-neutral format. The API has been tested on Intel, IBM and ARM architectures, and can be used to easily integrate with Variorum (see *Integrating with Variorum*).

Obtaining Power Consumption

The API to obtain node power has the following format. It takes a string (`char**`) by reference as input, and populates this string with a JSON object with CPU, memory, GPU (when available), and total node power. The total node power is estimated as a summation of available domains if it is not directly reported by the underlying architecture (such as Intel).

The `variorum_get_node_power_json(char**)` includes a string type JSON object with the following keys:

- hostname (string value)
- timestamp (integer value)
- power_node (real value)
- power_cpu_watts_socket* (real value)
- power_mem_watts_socket* (real value)

- power_gpu_watts_socket* (real value)

The "*" here refers to Socket ID. While more than one socket is supported, our test systems had only 2 sockets. Note that on the IBM Power9 platform, only the first socket (Chip-0) has the PWRSYS sensor, which directly reports total node power. Additionally, both sockets here report CPU, Memory and GPU power.

On Intel microarchitectures, total node power is not reported by hardware. As a result, total node power is estimated by adding CPU and DRAM power on both sockets.

For GPU power, IBM Power9 reports a single value, which is the sum of power consumed by all the GPUs on a particular socket. Our JSON object captures this with a `power_gpu_socket_*` interface, and does not report individual GPU power in the JSON object (this data is however available separately without JSON).

On systems without GPUs, or systems without memory power information, the value of the JSON fields is currently set to -1.0 to indicate that the GPU power or memory power cannot be measured directly. This has been done to ensure that the JSON object in itself stays vendor-neutral from a tools perspective. A future extension through NVML integration will allow for this information to report individual GPU power as well as total GPU power per socket with a cross-architectural build, similar to Variorum's `variorum_get_node_power()` API.

Querying Power Domains

The API for querying power domains allows users to query Variorum to obtain information about domains that can be measured and controlled on a certain architecture. It also includes information on the units of measurement and control, as well as information on the minimum and maximum values for setting the controls (control_range). If a certain domain is unsupported, it is marked as such.

The query API, `variorum_get_node_power_domain_info_json(char**)`, accepts a string by reference and includes the following vendor-neutral keys:

- hostname (string value)
- timestamp (integer value)
- measurement (comma-separated string value)
- control (comma-separated string value)
- unsupported (comma-separated string value)
- measurement_units (comma-separated string value)
- control_units (comma-separated string value)
- control_range (comma-separated string value)

Obtaining Node Utilization

The API to obtain node utilization has the following format. It takes a string (`char**`) by reference as input, and populates this string with a JSON object with total CPU, system CPU, user CPU, total memory, and GPU (when available) utilizations. It reports the utilization of each available GPU. GPU utilization is accomplished using the `int variorum_get_gpu_utilization_json(char **get_gpu_util_obj_str)` function. The total memory utilization is computed using `/proc/meminfo`, and CPU utilizations is computed using `/proc/stat`.

The `variorum_get_node_utilization_json(char **get_util_obj_str)` function returns a string type nested JSON object. An example is provided below:

```
{
    "hostname": {
        "CPU": {
```

(continues on next page)

(continued from previous page)

```
            "total_util%": (Real),
            "user_util%": (Real),
            "system_util%": (Real),
        },
        "memory_util%": (Real),
        "timestamp": (Integer),
        "GPU": {
            "Socket_*": {
                "GPUn*#_util%": (Integer)
            }
        }
    }
}
```

The * here refers to socket ID, and the # refers to GPU ID.

The `variorum_get_node_utilization_json(char **get_util_obj_str)` function returns a string type nested JSON object. An example is provided below:

```
{
    "hostname": {
        "timestamp": (Integer),
        "GPU": {
            "Socket_*": {
                "GPUn*#_util%": (Integer)
            }
        }
    }
}
```

The * here refers to socket ID, and the # refers to GPU ID.

5.4.4 Best Effort Power Capping

We support setting best effort node power limits in a vendor-neutral manner. This interface has been developed from the point of view of higher-level tools that utilize Variorum on diverse architectures and need to make node-level decisions. When the underlying hardware does not directly support a node-level power cap, a best-effort power cap is determined in software to provide an easier interface for higher-level tools (e.g. Flux, Kokkos, etc).

For example, while IBM Witherspoon inherently provides the ability to set a node-level power cap in watts in hardware through its OPAL infrastructure, Intel architectures currently do not support a direct node level power cap through MSRs. Instead, on Intel architectures, fine-grained CPU and DRAM level power caps can be dialed in using MSRs. Note that IBM Witherspoon does not provide fine-grained capping for CPU and DRAM level, but allows for a power-shifting ratio between the CPU and GPU components on a socket (see IBM documentation).

Our API, `variorum_cap_best_effort_node_power_limit()`, allows us to set a best effort power cap on Intel architectures by taking the input power cap value, and uniformly distributing it across sockets as CPU power caps. Currently, we do not set memory power caps, but we plan to develop better techniques for best-effort software capping in the future.

5.5 Example Programs

Variorum provides some examples in the `examples/` directory. These include examples of our APIs, usage with MPI and OpenMP, and an example for integrating with Variorum using the JSON API. Note that on Intel systems, we make a call to Variorum print API twice in our examples, as Intel systems require a delta between values to report adequate power numbers.

Note: All example codes have a print and print_verbose version showcasing the different printouts supported by Variorum.

5.5.1 Print Power Limit

The example below gets the power limits of the platform. The `print` API prints the output in tabular format that can be filtered and parsed by a data analysis framework, such as `R` or `Python`.

On an Intel platform, the output of this example should be similar to the following.

```
_LOG_VARIORUM_ENTER:~/variorum/src/variorum/variorum.c:variorum_print_power_limit::143
_PACKAGE_POWER_LIMITS Offset Host Socket Bits PowerLimit1_W TimeWindow1_sec PowerLimit2_
↪W TimeWindow2_sec
_PACKAGE_POWER_LIMITS 0x610 thompson 0 0x7851000158438 135.000000 1.000000 162.000000 0.
↪007812
_PACKAGE_POWER_LIMITS 0x610 thompson 1 0x7851000158438 135.000000 1.000000 162.000000 0.
↪007812
_DRAM_POWER_LIMIT Offset Host Socket Bits PowerLimit_W TimeWindow_sec
_DRAM_POWER_LIMIT 0x618 thompson 0 0x0 0.000000 0.000977
_DRAM_POWER_LIMIT 0x618 thompson 1 0x0 0.000000 0.000977
_PACKAGE_POWER_INFO Offset Host Socket Bits MaxPower_W MinPower_W MaxTimeWindow_sec␣
↪ThermPower_W
_PACKAGE_POWER_INFO 0x614 thompson 0 0x2f087001380438 270.000000 39.000000 40.000000 135.
↪000000
_PACKAGE_POWER_INFO 0x614 thompson 1 0x2f087001380438 270.000000 39.000000 40.000000 135.
↪000000
_RAPL_POWER_UNITS Offset Host Socket Bits PowerUnit_W EnergyUnit_J TimeUnit_sec
_RAPL_POWER_UNITS 0x606 thompson 0 0xa0e03 0.125000 0.000061 0.000977
_RAPL_POWER_UNITS 0x606 thompson 1 0xa0e03 0.125000 0.000061 0.000977
_LOG_VARIORUM_EXIT:~/variorum/src/variorum/variorum.c:variorum_print_power_limit::165
```

On an IBM Power9 platform, the output may look similar to:

```
_LOG_VARIORUM_ENTER:~/variorum/src/variorum/variorum.c:variorum_print_power_limit::143
_POWERCAP Host CurrentPower_W MaxPower_W MinPower_W PSR_CPU_to_GPU_0_% PSR_CPU_to_GPU_8_%
_POWERCAP lassen3 3050 3050 500 100 100
_LOG_VARIORUM_EXIT:~/variorum/src/variorum/variorum.c:variorum_print_power_limit::165

On an Nvidia GPU platform, the output may look similar to:
```

```
_LOG_VARIORUM_ENTER:~/variorum/src/variorum/variorum.c:variorum_print_power_limit::143
_GPU_POWER_LIMIT Host Socket DeviceID PowerLimit_W
_GPU_POWER_LIMIT lassen1 0 0 300.000
_GPU_POWER_LIMIT lassen1 0 1 300.000
```

(continues on next page)

(continued from previous page)

```
_GPU_POWER_LIMIT lassen1 1 2 300.000
_GPU_POWER_LIMIT lassen1 1 3 300.000
_LOG_VARIORUM_EXIT:~/variorum/src/variorum/variorum.c:variorum_print_power_limit::165
```

5.5.2 Print Verbose Power Limit

The example below gets the power limits of the platform. The `print_verbose` API prints the output in verbose format that is more human-readable (with units, titles, etc.).

```c
// Copyright 2019-2023 Lawrence Livermore National Security, LLC and other
// Variorum Project Developers. See the top-level LICENSE file for details.
//
// SPDX-License-Identifier: MIT

#include <getopt.h>
#include <stdio.h>

#include <variorum.h>

int main(int argc, char **argv)
{
    int ret;

    const char *usage = "Usage: %s [-h] [-v]\n";
    int opt;
    while ((opt = getopt(argc, argv, "hv")) != -1)
    {
        switch (opt)
        {
            case 'h':
                printf(usage, argv[0]);
                return 0;
            case 'v':
                printf("%s\n", variorum_get_current_version());
                return 0;
            default:
                fprintf(stderr, usage, argv[0]);
                return -1;
        }
    }

    ret = variorum_print_verbose_power_limit();
    if (ret != 0)
    {
        printf("Print verbose power limit failed!\n");
    }
    return ret;
}
```

On an Intel platform, the output of this example should be similar to the following:

```
_LOG_VARIORUM_ENTER:~/variorum/src/variorum/variorum.c:variorum_print_verbose_power_
↪limit::180
_PACKAGE_POWER_LIMIT Offset: 0x610, Host: thompson, Socket: 0, Bits: 0x7851000158438,␣
↪PowerLimit1: 135.000000 W, TimeWindow1: 1.000000 sec, PowerLimit2: 162.000000 W,␣
↪TimeWindow2: 0.007812 sec
_PACKAGE_POWER_LIMIT Offset: 0x610, Host: thompson, Socket: 1, Bits: 0x7851000158438,␣
↪PowerLimit1: 135.000000 W, TimeWindow1: 1.000000 sec, PowerLimit2: 162.000000 W,␣
↪TimeWindow2: 0.007812 sec
_DRAM_POWER_LIMIT Offset: 0x618, Host: thompson, Socket: 0, Bits: 0x0, PowerLimit: 0.
↪000000 W, TimeWindow: 0.000977 sec
_DRAM_POWER_LIMIT Offset: 0x618, Host: thompson, Socket: 1, Bits: 0x0, PowerLimit: 0.
↪000000 W, TimeWindow: 0.000977 sec
_PACKAGE_POWER_INFO Offset: 0x614, Host: thompson, Socket: 0, Bits: 0x2f087001380438,␣
↪MaxPower: 270.000000 W, MinPower: 39.000000 W, MaxWindow: 40.000000 sec, ThermPower:␣
↪135.000000 W
_PACKAGE_POWER_INFO Offset: 0x614, Host: thompson, Socket: 1, Bits: 0x2f087001380438,␣
↪MaxPower: 270.000000 W, MinPower: 39.000000 W, MaxTimeWindow: 40.000000 sec,␣
↪ThermPower: 135.000000 W
_RAPL_POWER_UNITS Offset: 0x606, Host: thompson, Socket: 0, Bits: 0xa0e03, PowerUnit: 0.
↪125000 W, EnergyUnit: 0.000061 J, TimeUnit: 0.000977 sec
_RAPL_POWER_UNITS Offset: 0x606, Host: thompson, Socket: 1, Bits: 0xa0e03, PowerUnit: 0.
↪125000 W, EnergyUnit: 0.000061 J, TimeUnit: 0.000977 sec
_LOG_VARIORUM_EXIT:~/variorum/src/variorum/variorum.c:variorum_print_verbose_power_
↪limit::202
```

On an IBM platform, the output may look similar to:

```
_LOG_VARIORUM_ENTER:~/variorum/src/variorum/variorum.c:variorum_print_verbose_power_
↪limit::180
_POWERCAP Host: lassen3, CurrentPower: 3050 W, MaxPower: 3050 W, MinPower: 500 W, PSR_
↪CPU_to_GPU_0: 100%, PSR_CPU_to_GPU_8: 100%
_LOG_VARIORUM_EXIT:~/variorum/src/variorum/variorum.c:variorum_print_verbose_power_
↪limit::202
```

On an Nvidia platform, the output may look similar to:

```
_LOG_VARIORUM_ENTER:~/variorum/src/variorum/variorum.c:variorum_print_verbose_power_
↪limit::180
_GPU_POWER_LIMIT Host: lassen1, Socket: 0, DeviceID: 0, PowerLimit: 300.000 W
_GPU_POWER_LIMIT Host: lassen1, Socket: 0, DeviceID: 1, PowerLimit: 300.000 W
_GPU_POWER_LIMIT Host: lassen1, Socket: 1, DeviceID: 2, PowerLimit: 300.000 W
_GPU_POWER_LIMIT Host: lassen1, Socket: 1, DeviceID: 3, PowerLimit: 300.000 W
_LOG_VARIORUM_EXIT:~/variorum/src/variorum/variorum.c:variorum_print_verbose_power_
↪limit::202
```

5.6 Supported Platform Documentation

These are the currently supported platforms in Variorum.

5.6.1 AMD Overview

AMD platforms support in-band monitoring and control through sensors and machine-specific registers for CPUs as well as GPUs. AMD provides and open-source stack of its drivers as well as its in-band libraries that Variorum leverages.

Requirements for AMD CPUs

Beginning with Variorum 0.5.0, AMD processors from the AMD EPYC Milan family 19h, models 0-Fh and 30h-3Fh are supported. The current port is also expected to be supported on the upcoming AMD EPYC Genoa architecture. This functionality has been tested on Linux distributions SLES15 and Ubuntu 18.04. This port depends on the AMD open-sourced software stack components listed below:

1. EPYC System Management Interface In-band Library (E-SMI library) available at https://github.com/amd/esmi_ib_library
2. AMD Energy Driver https://github.com/amd/amd_energy
3. HSMP driver for power metrics https://github.com/amd/amd_hsmp

The E-SMI library provides the C API for user space application of the AMD Energy Driver and the AMD HSMP modules.

The AMD Energy Driver is an out-of-tree kernel module that allows for core and socket energy counter access through MSRs and RAPL via `hwmon` sys entries. These registers are updated every millisecond and cleared on reset of the system. Some registers of interest include:

- **Power, Energy and Time Units**
  - `MSR_RAPL_POWER_UNIT/ C001_0299`: shared with all cores in the socket
- **Energy consumed by each core**
  - `MSR_CORE_ENERGY_STATUS/ C001_029A`: 32-bitRO, Accumulator, core-level power reporting
- **Energy consumed by Socket**
  - `MSR_PACKAGE_ENERGY_STATUS/ C001_029B`: 32-bitRO, Accumulator, socket-level power reporting, shared with all cores in socket

The Host System Management Port (HSMP) kernel module allows for *setting* of power caps, boostlimits and PCIe access. It provides user level access to the HSMP mailboxes implemented by the firmware in the System Management Unit (SMU). AMD Power Control Knobs are exposed through HSMP via sysfs.

- **`amd_hsmp/cpuX/`: Directory for each possible CPU**
  - `boost_limit` (RW): HSMP boost limit for the core in MHz
- **`amd_hsmp/socketX/`: Directory for each possible socket**
  - `boost_limit` (WO): Set HSMP boost limit for the socket in MHz
  - `c0_residency` (RO): Average % all cores are in C0 state
  - `cclk_limit` (RO): Most restrictive core clock (CCLK) limit in MHz
  - `fabric_clocks` (RO): Data fabric (FCLK) and memory (MCLK) in MHz

  - `fabric_pstate` (WO): Set data fabric P-state, -1 for autonomous
  - `power` (RO): Average socket power in milliwatts
  - `power_limit` (RW): Socket power limit in milliwatts
  - `power_limit_max` (RO): Maximum possible value for power limit in mW
  - `proc_hot` (RO): Socket PROC_HOT status (1 = active, 0 = inactive)
  - `tctl` (RO): Thermal Control value (not temperature)

We expect a similar software stack to be available on the upcoming El Capitan supercomputer at Lawrence Livermore National Laboratory.

Requirements for AMD GPUs

Beginning with Variorum 0.6.0, we support AMD Radeon Instinct GPUs with the help of the Radeon Open Compute management (ROCm) stack. The Variorum AMD GPU port currently requires ROCm System Management Interface (ROCm-SMI) v5.2.0, and supports various AMD GPUs including (but not limited) to MI50, MI60, MI100, and MI200. Future versions on ROCm-SMI are expected to be backward compatible, and upcoming AMD GPU hardware for El Capitan supercomputer is expected to be supported through ROCm-SMI as well.

Monitoring and Control Through E-SMI API

Variorum interfaces with AMD's E-SMI library for obtaining power and energy information. These E-SMI APIs are described below.

The built-in monitoring interface on the AMD EPYC™ processors is implemented by the SMU FW. All registers are updated every 1 millisecond.

Power telemetry

- `esmi_socket_power_get()`: Instantaneous power is reported in milliwatts
- `esmi_socket_power_cap_get()` and `esmi_socket_power_cap_set()`: Get and Set power limit of the socket in milliwatts
- `esmi_socket_power_cap_max_get()`: Maximum Power limit of the socket in milliwatts

Boostlimit telemetry

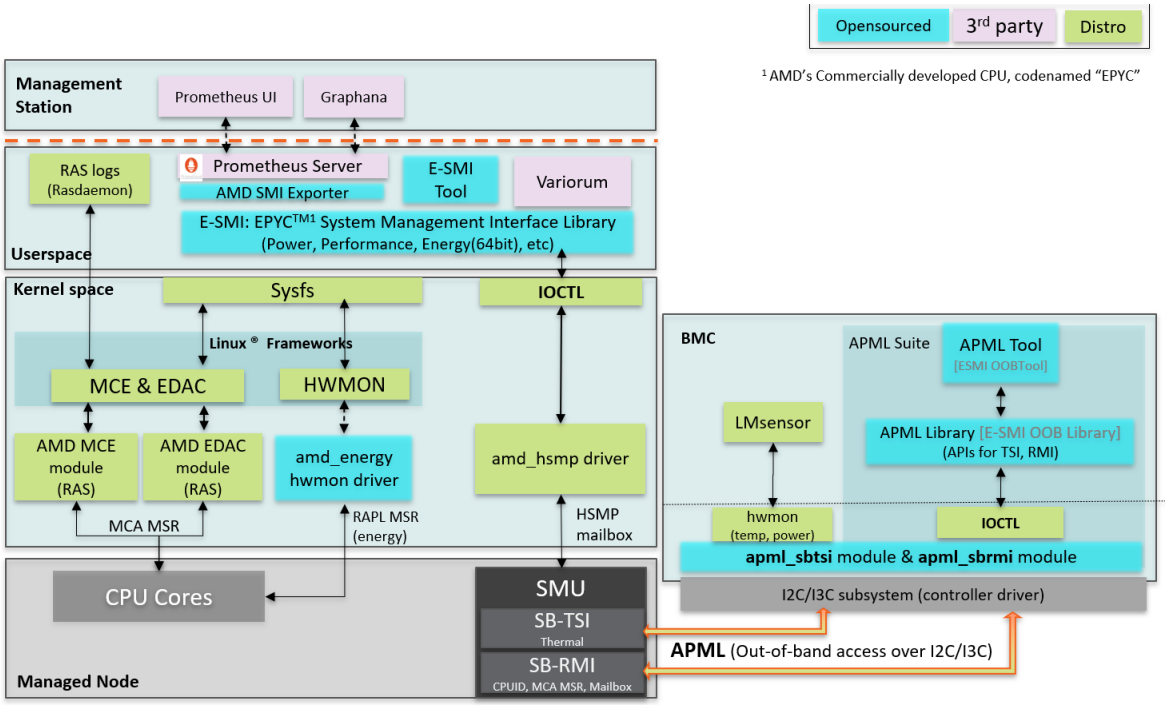
Boostlimit is a maximum frequency a CPU can run at.

- `esmi_core_boostlimit_get()` and `esmi_core_boostlimit_set()`: Get and set the current boostlimit for a given core
- `esmi_socket_boostlimit_set()`: Set boostlimit for all the cores in the socket

Energy telemetry

- `esmi_socket_energy_get()`: Get software accumulated 64-bit energy counter for a given socket
- `esmi_core_energy_get()`: Get software accumulated 64-bit energy counter for a given core

Details of the AMD E-SMS CPU stack can be found on the AMD Developer website. We reproduce a figure from this stack below.



Monitoring and Control Through ROCM-SMI API

Variorum interfaces with AMD's ROCm-SMI library for obtaining power and energy information for GPUs. These ROCm-SMI APIs are described below.

- `rsmi_num_monitor_devices`: Get the number of GPU devices.
- `rsmi_dev_power_ave_get`: Get the current average power consumption of a GPU device over a short time window in microwatts.
- `rsmi_dev_power_cap_get`: Get the current power cap in microwatts on a GPU device which, when reached, causes the system to take action to reduce power.
- `rsmi_dev_power_cap_range_get`: Get the range of valid values for the power cap, including the maximum possible and the minimum possible cap on a GPU device.
- `rsmi_dev_temp_metric_get`: Get the temperature metric value for the specified metric and sensor (e.g. Current or Max temperature), from the GPU device.
- `rsmi_dev_gpu_clk_freq_get`: Get the list of possible system clock speeds for a GPU device for a specified clock type (e.g. Graphics or Memory clock).
- `rsmi_utilization_count_get`: Get coarse grain utilization counter of the specified GPU device, including graphics and memory activity counters.
- `rsmi_dev_power_cap_set`: Set the GPU device power cap for the specified GPU device in microwatts.

References

- AMD EPYC processors Fam19h technical reference
- AMD ROCm-SMI technical reference

5.6.2 ARM Overview

Variorum supports two flavors of ARM architectures: - Arm Juno r2 SoC - Ampere Neoverse N1

The Arm Juno r2 platform is a big.LITTLE cluster with Cortex-A72 (big) and Cortex-A53 (LITTLE) clusters (also called processors), respectively. It also has an Arm Mali GP-GPU. We have tested the ARM functionality of Variorum on Linaro supported Linux.

The Ampere Neoverse N1 platform is an 80-core single-processor platform with two Nvidia Ampere GPUs.

Requirements

This version of the ARM port of Variorum depends on the Linux Hardware Monitoring (hwmon) subsystem for access to the telemetry and control interfaces on the tested ARM platform. The standardized data interfaces provided by the hwmon framework enable a generic ARM implementation of Variorum.

Model Identification

Variorum use the *MIDR_EL1* (Main ID) register which provides the identification information of the ARM processor to initialize the low-level interfaces on the target architecture. Specifically, Variorum uses bits *[15:4]* of *MIDR_E1* to get the primary part number. For Neoverse N1 the primary part number is *0xD0C* whereas for Arm Juno r2 big.LITTLE SoC the primary part numbers are *0xD08* (Cortex-A72) and *0xD03* (Cortex-A53).

Monitoring and Control Through Sysfs Interface

The built-in monitoring interface on the Arm Juno r2 board is implemented by the on-board FPGA. Since this interface is not universally available on most Arm implementations, we leverage the standard Linux sysfs interface for monitoring and control. The following subsections provide the specific metrics that are monitored on Arm Juno r2:

Power telemetry

The sysfs interface provides a file for each of the following Advanced Peripheral Bus (APB) energy meter registers:

- SYS_POW_SYS : `/sys/class/hwmon/hwmon0/power1_input`
- SYS_POW_A72 : `/sys/class/hwmon/hwmon0/power2_input`
- SYS_POW_A53 : `/sys/class/hwmon/hwmon0/power3_input`
- SYS_POW_GPU : `/sys/class/hwmon/hwmon0/power4_input`

Instantaneous power is recorded in bits 0-23 and reported in microwatts by the sysfs interface. To improve readability of the verbose output Variorum converts power into milliwatts before reporting. All registers are updated every 100 microseconds.

Memory power telemetry is not available on this platform.

Thermal telemetry

The sysfs interface provides a file for thermal telemetry for each system component through the GetSensorValue command in CSS System Control and Power Interface (SCPI).

- SoC temperature: `/sys/class/hwmon/hwmon0/temp1_input`
- big temperature: `/sys/class/hwmon/hwmon0/temp2_input`
- LITTLE temperature: `/sys/class/hwmon/hwmon0/temp3_input`
- GPU temperature: `/sys/class/hwmon/hwmon0/temp4_input`

Instantaneous temperatures are reported in degree Celsius.

On the Neoverse N1 system, the following thermal sensors are provided:

- Ethernet connector temperature: `/sys/class/hwmon/hwmon0/temp1_input`
- SoC temperature: `/sys/class/hwmon/hwmon1/temp1_input`

Clocks telemetry

Clocks are collected by the sysfs interface using the GetClockValue command in SCPI on both of the supported ARM platforms. On the Arm Juno r2 platform, a separate `policy*/` subdirectory is provided for the big and LITTLE clusters.

- big clocks: `/sys/devices/system/cpu/cpufreq/policy0/scaling_cur_freq`
- LITTLE clocks: `/sys/devices/system/cpu/cpufreq/policy1/scaling_cur_freq`

On the Neoverse N1 platform, a separate `policy*/` subdirectory is provided for each core in the SoC.

- Core clocks: `/sys/devices/system/cpu/cpufreq/policy[0-79]/scaling_cur_freq`

Frequencies are reported by the sysfs interface in KHz. Variorum reports the clocks in MHz to keep it consistent with the clocks reported for other supported architectures.

Frequency control

The sysfs interface uses the SetClockValue SCPI command to set processor frequency for the following user-facing interfaces on the supported platforms:

Arm Juno r2:

- big clocks: `/sys/devices/system/cpu/cpufreq/policy0/scaling_setspeed`
- LITTLE clocks: `/sys/devices/system/cpu/cpufreq/policy1/scaling_setspeed`

Neoverse N1:

- core clocks: `/sys/devices/system/cpu/cpufreq/policy[0-79]/scaling_setspeed`

New frequency is specified in KHz to these interfaces. Variorum takes the new frequency as input in MHz and performs this conversion internally.

If you run into an error accessing the sysfs interface, this could be due to an the specified frequency value or the set governor. The sysfs interface only accepts valid values for frequencies as output by `policy*/scaling_available_frequencies`. Also, the specified frequency is only applied when the governor in `policy*/scaling_governor` is set to *userspace*.

References

- Arm Juno r2 technical reference
- Neoverse N1 technical reference
- Cortex-A53 technical reference
- Cortex-A72 technical reference
- hwmon sysfs interface
- hwmon documentation
- Energy Monitoring on Juno

5.6.3 IBM Power9 Overview

IBM Power9 architecture supports in band monitoring with sensors and out of band power capping with OPAL. These depend on specific IBM files that we describe below. Permissions on these files can be modified through cgroups. OPAL/Skiboot is part of IBM provided firmware that is expected to be present on the system.

Requirements

Read access to `/sys/firmware/opal/exports/occ_inband_sensors` is required, along with read-write access to `/sys/firmware/opal/powercap/system_powercap/powercap_current` and `/sys/firmware/opal/psr/`. This can be enabled by using group permissions. For example, to allow only users belonging to certain group to set the power cap or power shifting ratio, `udev` can be used as follows.

```
$ cat /etc/udev/rules.d/99-coral.rules

KERNELS=="*", ACTION=="*", DEVPATH=="/devices/*", RUN+="/bin/chown root:coral
    /sys/firmware/opal/powercap/system-powercap/powercap-current
    /sys/firmware/opal/psr/cpu_to_gpu_0
    /sys/firmware/opal/psr/cpu_to_gpu_8"
```

The above file needs to be copied to all nodes. The administrator has to create a group (for example, named `coral` below) and add the users to this group. The `udev` rule can then be set as follows:

```
$ udevadm trigger /sys/block/sda

$ ls -l /sys/firmware/opal/powercap/system-powercap/powercap-current \
/sys/firmware/opal/psr/cpu_to_gpu_0 /sys/firmware/opal/psr/cpu_to_gpu_8

-rw-rw-r-- 1 root coral 65536 Jul  3 06:19 /sys/firmware/opal/powercap/system-powercap/
↪powercap-current
-rw-rw-r-- 1 root coral 65536 Jul  3 06:19 /sys/firmware/opal/psr/cpu_to_gpu_0
-rw-rw-r-- 1 root coral 65536 Jul  3 06:19 /sys/firmware/opal/psr/cpu_to_gpu_8
```

Inband Sensors for Monitoring

The OCC (On-Chip-Controller) periodically reads various sensors related to power, temperature, CPU frequency, CPU utilization, memory bandwidth, etc. The sensor data is stored in OCC's SRAM and is available to the user inband through the sensors file listed below:

- Key file for inband sensors: `/sys/firmware/opal/exports/occ_inband_sensors`

OCC Sensor Data formatting is described below, and we then describe the code structures that were used to represent this data in the IBM port of Variorum.

OCC Sensor Data

OCC sensor data will use BAR2 (OCC Common is per physical drawer). Starting address is at offset 0x00580000 from BAR2 base address. Maximum size is 1.5MB.

Start (Offset from BAR2 base address)	End	Size	Description
0x00580000	0x005A57FF	150kB	OCC 0 Sensor Data Block
0x005A5800	0x005CAFFF	150kB	OCC 1 Sensor Data Block
:	:	:	:
0x00686800	0x006ABFFF	150kB	OCC 7 Sensor Data Block
0x006AC000	0x006FFFFF	336kB	Reserved

OCC N Sensor Data Block Layout (150kB)

The sensor data block layout is the same for each OCC N. It contains sensor-header-block, sensor-names buffer, sensor-readings-ping buffer and sensor-readings-pong buffer.

Start (Offset from OCC N Sensor Data Block)	End	Size	Description
0x00000000	0x000003FF	1kB	Sensor Data Header Block
0x00000400	0x0000CBFF	50kB	Sensor Names
0x0000CC00	0x0000DBFF	4kB	Reserved
0x0000DC00	0x00017BFF	40kB	Sensor Readings ping buffer
0x00017C00	0x00018BFF	4kB	Reserved
0x00018C00	0x00022BFF	40kB	Sensor Readings pong buffer
0x00022C00	0x000257FF	11kB	Reserved

There are eight OCC Sensor Data Blocks. Each of these has the same data block layout. Within each sensor data block, we have:

- **data header block**: Written once at initialization, captured in occ_sensor_data_header struct (reading_version in this struct defines the format of the ping/pong buffer, this could be READING_FULL or READING_COUNTER).
- **names block**: Written once at initialization, captured in occ_sensors_name
- **readings ping buffer and readings pong buffer**: The ping/pong buffers are two 40kB buffers, one is being updated by the OCC and the other is available for reading. Both have the same format version (defined in sensor_struct_type and struct_attr).

There are four enums:

1. **occ_sensor_type**: e.g., CURRENT, VOLTAGE, TEMPERATURE, POWER, etc.

2. **occ_sensor_location**: e.g., SYSTEM, PROCESSOR, MEMORY, CORE, etc.
3. **sensor_struct_type**: READING_FULL, READING_COUNTER (ties to reading_version)
4. **sensor_attr**: SAMPLE, ACCUMULATOR (also has to do with reading_version)

There are four structs:

1. **occ_sensor_data_header**: Gives us offsets to ping and pong buffers, format version of the ping and pong buffers (reading_version), and offset to location of the names buffer.
2. **occ_sensor_name**: Format of the sensor. Gives us the type of sensor, location of sensor, name of sensor, unit of sensor, update frequency of sensor, format of ping/pong buffer of that particular sensor, offset for reading buffers for this particular sensor.
3. **occ_sensor_record**: This is the data if you were using READING_FULL. Contains timestamp, latest sample or latest accumulated value, min and max values for sample, job scheduler, profiler and CSM (we're not clear about what these are). We think the sample one is the interesting one for our purpose at the moment.
4. **occ_sensor_counter**: This is the data if you were using READING_COUNTER. Contains timestamp, latest sample or latest accumulated value. unit_8 values and no min/max values are reported here.

Inband Power Capping and GPU Shifting Ratio

Power caps and GPU power shifting ratio can be set by using OPAL/Skiboot. This is an inband interface through the BMC located on the node.

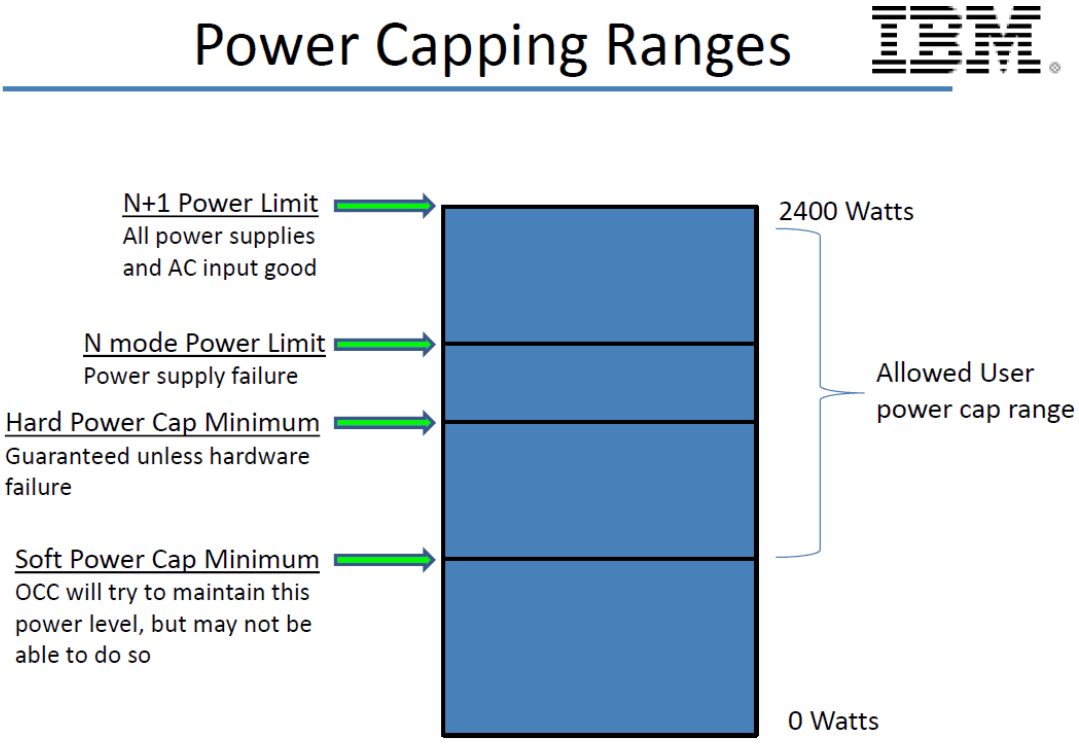
Node power caps are set by writing to the following file in Watts: `/sys/firmware/opal/powercap/system-powercap/powercap-current`

Socket level power capping and memory power capping is not available.

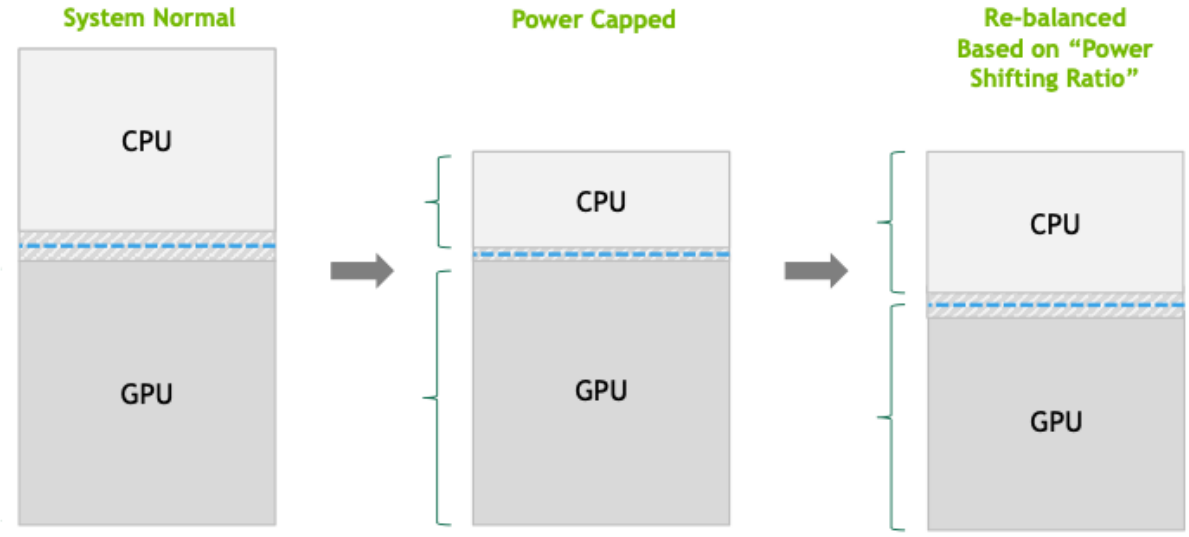
GPU power shifting ratio can be set by setting the following files in percentage (i.e., between 0 and 100). `/sys/firmware/opal/psr/cpu_to_gpu_0` and `/sys/firmware/opal/psr/cpu_to_gpu_8`

Write access to these files is needed to set node power caps and GPU ratio.

The figure below depicts the ranges for IBM power caps on Power9 system (reproduced with permission from our IBM collaborators).



The figure below shows the details of GPU power shifting ratio.



References

- OCC
- OPAL
- Skiboot
- Inband Sensors

5.6.4 Intel Overview

Intel processors have the most sophisticated power and thermal control of any processor vendor we work with. While Intel's documentation remains the final arbiter, that format has not allowed the community of Intel users to discuss best practices and distribute documentation patches. For this release we provide below a listing of the MSRs found in Chapter 14 of volume 3B of Intel's SDM, plus a few related MSRs that exist elsewhere in public documentation. Alongside the register diagrams we note what we have learned (if anything) by using the registers and discussing them with our colleagues at Intel and elsewhere.

Requirements

To use Variorum on Intel platforms, access to low-level registers needs to be enabled for non-root users. This can be enabled with the msr-safe kernel driver which must be loaded to enable user-level read and write of allowed MSRs.

The msr-safe driver provides the following device files:

```
/dev/cpu/<CPU>/msr_safe
```

Alternately, Variorum can be used as root with the stock MSR kernel driver loaded.

```
modprobe msr
```

The kernel driver provides an interface to read and write MSRs on an x86 processor.

The stock MSR driver provides the following device files:

```
ls /dev/cpu/<CPU>/msr
```

Best Practices

These are the most common mistakes we have seen when using these registers.

- **IA32_PERF_CTL does not set clock frequency**
  - In the distant past prior to multicore and turbo, setting IA32_PERF_CTL might have had the effect of dialing in the requested CPU clock frequency. In any processor since Nehalem, however, it sets a frequency cap.
- **Always measure effective clock frequency using IA32_APERF and IA32_MPERF.**
  - Given the amount of performance variation within the operating system and within and across processors, it is easy to talk oneself into a story of how a particular dial relates to performance by changing the clock frequency. Measuring both execution time and clock frequency (and perhaps IPC as well) is an excellent filter for those stories.
- Do not use Linux performance governors as they have limited support.

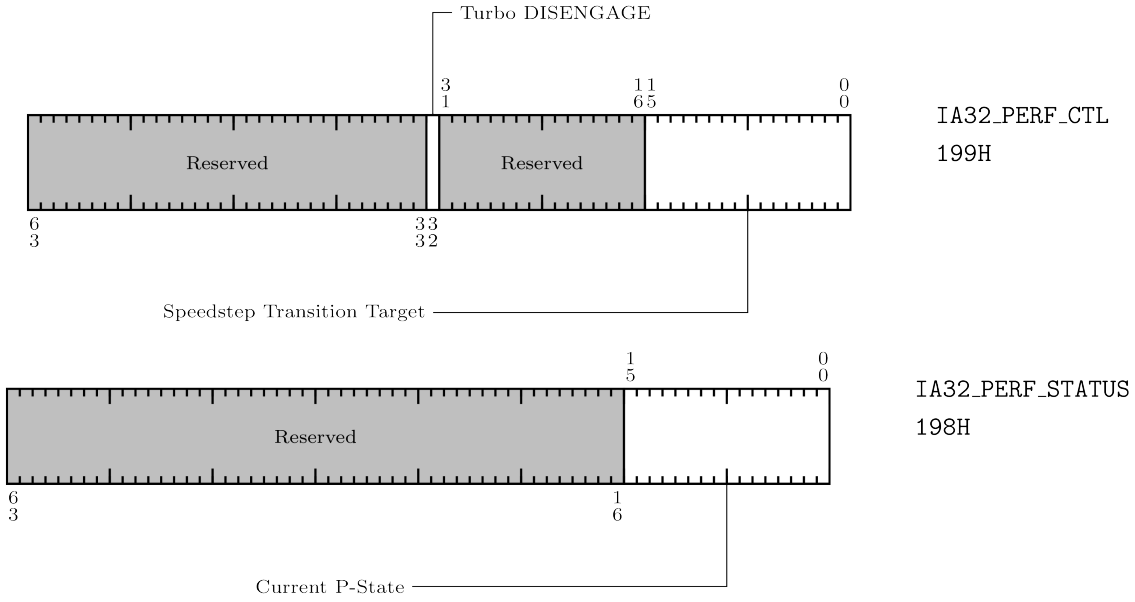
- **Not all encodable values are effective.**
  - The canonical case here is RAPL time windows. There is a minimum value supported in firmware, and any request for less than that minimum is silently clipped.

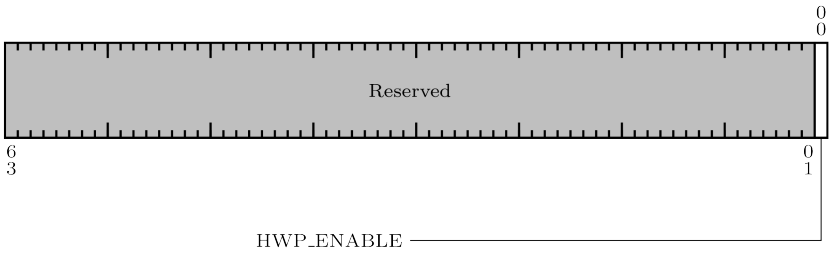
Caveats

- **Intel naming conventions are often inconsistent.**
  - Naming conventions will vary across and within documents, even to the naming of particular MSRs. While these are trivial to the eye (CTL versus CONTROL, PKG versus PACKAGE) it does make grepping documents more challenging than it should be. We have tried to follow a consistent scheme for MSRs, PCI addresses and CPUID queries. Where there is a conflict in MSR names, we have chosen what seems most sensible.
- **Determining which MSRs are available on which processors is problematic.**
  - Motherboard manufacturers can mask out available MSRs, and Intel's documentation can contain errors.

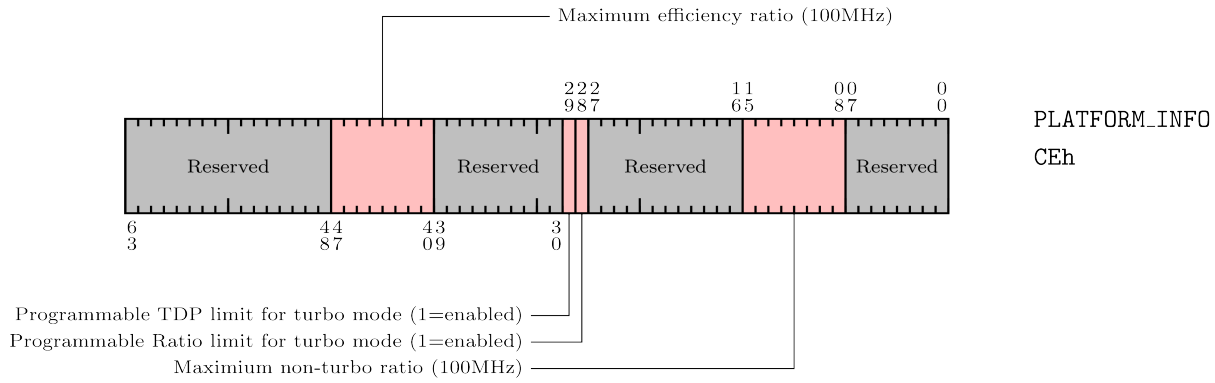
Enhanced Intel Speedstep Technology

- Exists if CPUID.(EAX=1):ECX[7] == 1.
- Enabled by IA32_MISC_ENABLE[16] <- 1.
- **MSRs used:**
  - IA32_PERF_CTL
  - IA32_PERF_STATUS
  - IA32_PM_ENABLE
  - MSR_PLATFORM_INFO





- IA32_PM_ENABLE will disable IA32_PERF_CTL. The enable bit is sticky and requires a reset to clear.



- MSR_PLATFORM_INFO Maximum Efficiency Ratio is the only guaranteed frequency regardless of workload.

P-State Hardware Coordination

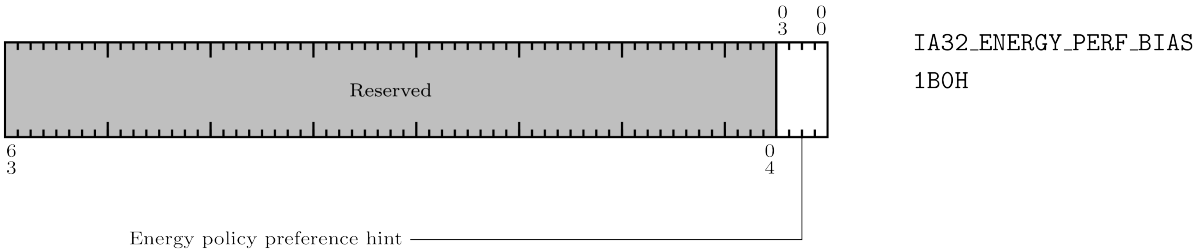
- Exists if CPUID.(EAX=6):ECX[0] == 1
- **MSRs used:**
  - IA32_MPERF
  - IA32_APERF

Intel Dynamic Acceleration Technology/Intel Turbo Boost Technology

- Enabled by MSR_MISC_ENABLE[38] <- 1, IA32_PERF_CTL[32] <- 0
- Note that the former is intended for one-time use by BIOS, the latter is intended for dynamic control.

Performance and Energy Bias Hint Support

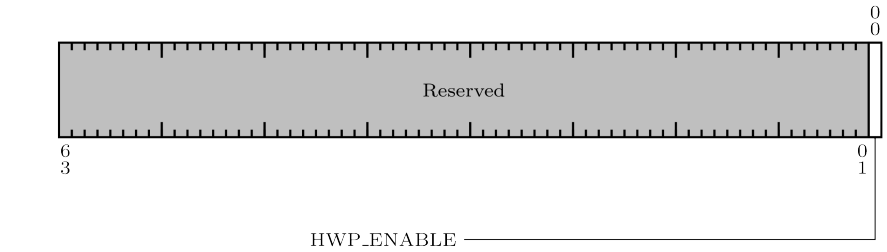
- Exists if CPUID.(EAX=6):ECX[3] == 1
- **MSRs used:**
  - IA32_ENERGY_PERF_BIAS



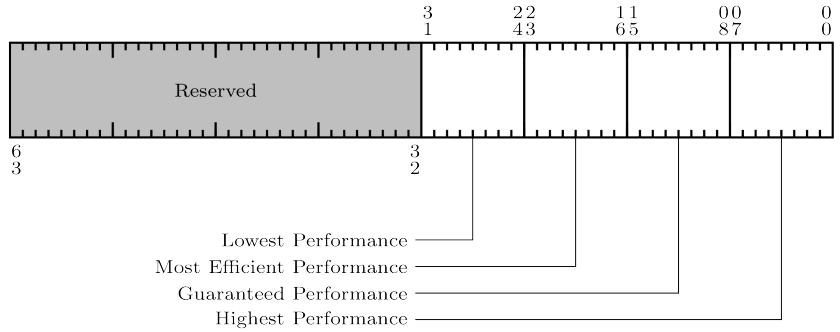
Hardware Controlled Performance States

- If CPUID.(EAX=6):EAX[7] == 1, then IA32_PM_ENABLE, IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS present.
- If CPUID.(EAX=6):EAX[8] == 1, then IA32_HWP_INTERRUPT present.
- If CPUID.(EAX=6):EAX[9] == 1, then IA32_HWP_REQUEST contains a programmable activity window.
- If CPUID.(EAX=6):EAX[10]== 1, then IA32_HWP_REQUEST has a programmable energy/performance hint.
- If CPUID.(EAX=6):EAX[11]== 1, then IA32_HWP_REQUEST_PKG is present.
- If CPUID.(EAX=6):EAX[20]== 1 and a single logical processor of a core is active, requests originating in the idle virtual processor via IA32_HWP_REQUEST_MSR are ignored.
- If CPUID.(EAX=6):EAX[18]== 1, IA32_HWP_REQUEST writes become visible outside the originating logical processor via “fast writes.”
- **MSRs used:**
  - IA32_PM_ENABLE
  - IA32_HWP_CAPABILITIES
  - IA32_HWP_REQUEST_PKG
  - IA32_HWP_INTERRUPT
  - IA32_HWP_REQUEST
  - IA32_HWP_PECI_REQUEST_INFO
  - IA32_HWP_STATUS
  - IA32_THERM_STATUS
  - MSR_PPERF
  - FAST_UNCORE_MSRS_CAPABILITY

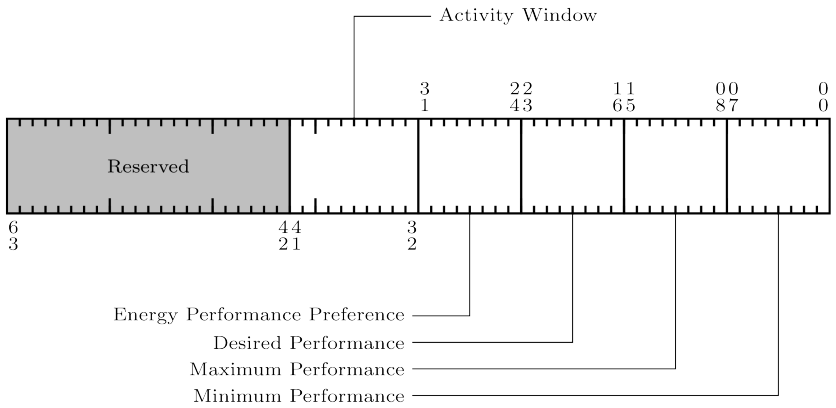
- FAST_UNCORE_MSRS_CTL
- FAST_UNCORE_MSRS_STATUS



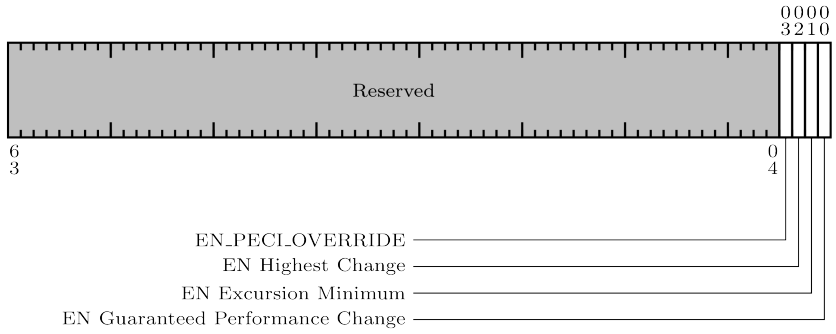
IA32_PM_ENABLE
770H



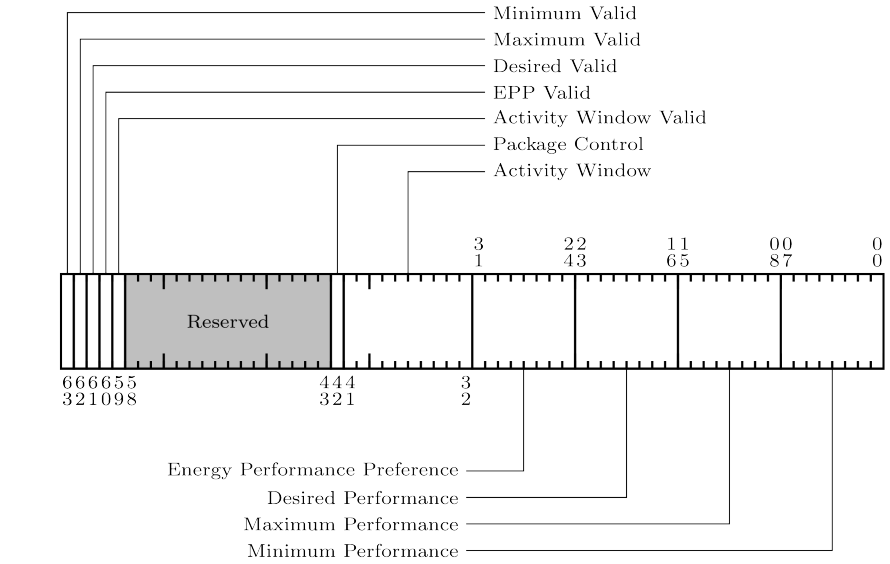
IA32_HWP_CAPABILITIES
771H



IA32_HWP_REQUEST_PKG
772H

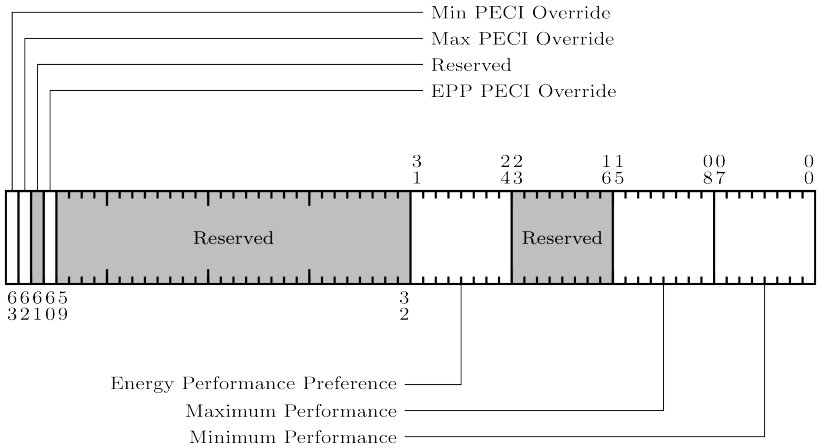


IA32_HWP_INTERRUPT
773H



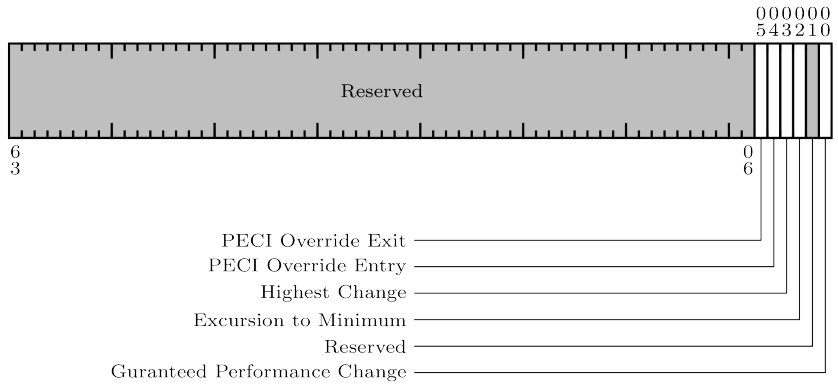
IA32_HWP_REQUEST

774H



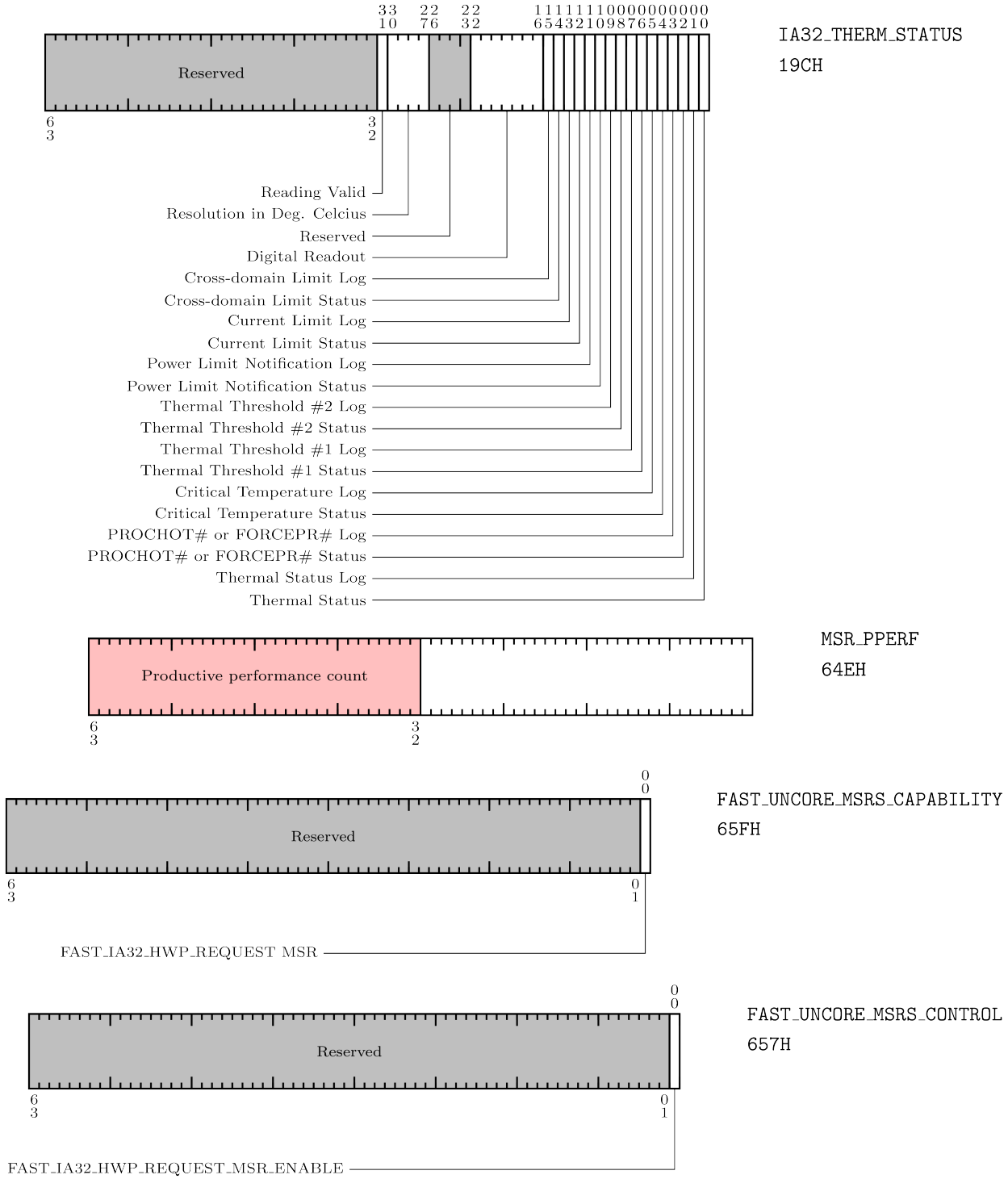
IA32_HWP_PECI_REQUEST_INFO

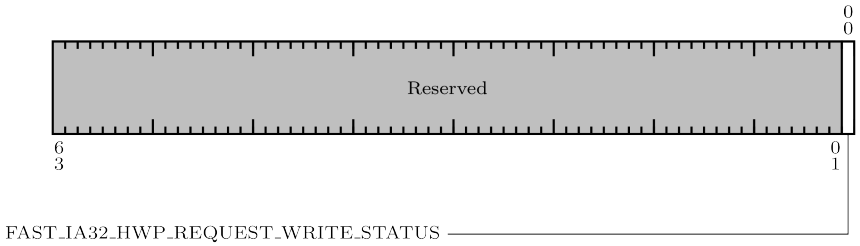
775H



IA32_HWP_STATUS

777H

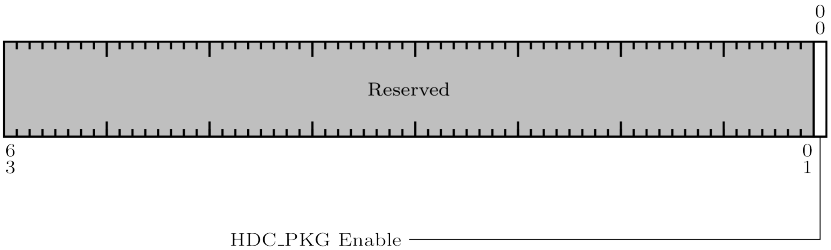

33 22 22 11111110000000000
10 76 32 65432109876543210
Reserved
63
32
Reading Valid
Resolution in Deg. Celcius
Reserved
Digital Readout
Cross-domain Limit Log
Cross-domain Limit Status
Current Limit Log
Current Limit Status
Power Limit Notification Log
Power Limit Notification Status
Thermal Threshold #2 Log
Thermal Threshold #2 Status
Thermal Threshold #1 Log
Thermal Threshold #1 Status
Critical Temperature Log
Critical Temperature Status
PROCHOT# or FORCEPR# Log
PROCHOT# or FORCEPR# Status
Thermal Status Log
Thermal Status
IA32_THERM_STATUS
19CH
Productive performance count
63
32
MSR_PPERF
64EH
00
Reserved
63
01
FAST_IA32_HWP_REQUEST MSR
FAST_UNCORE_MSRS_CAPABILITY
65FH
00
Reserved
63
01
FAST_IA32_HWP_REQUEST_MSR_ENABLE
FAST_UNCORE_MSRS_CONTROL
657H



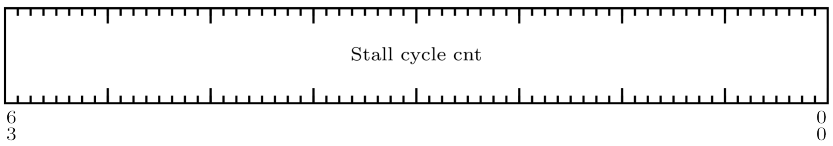
FAST_UNCORE_MSRS_STATUS
65EH

Hardware Duty Cycling

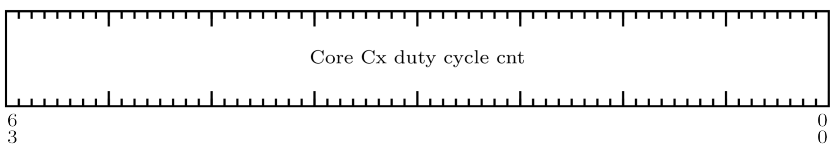
- Present if CPUID.(EAX=6):EAX[13] == 1
- **MSRs used:**
  - IA32_PKG_HDC_CTL
  - IA32_PM_CTL1
  - IA32_THREAD_STALL
  - MSR_CORE_HDC_RESIDENCY
  - MSR_PKG_HDC_SHALLOW_RESIDENCY
  - MSR_PKG_HDC_DEEP_RESIDENCY
  - MSR_PKG_HDC_CONFIG



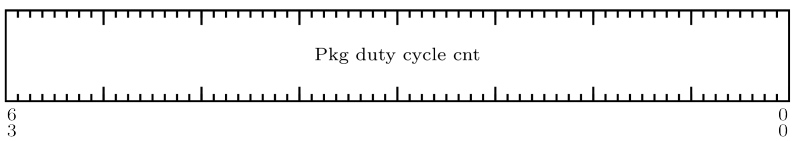
IA32_PKG_HDC_CTL
DB0H



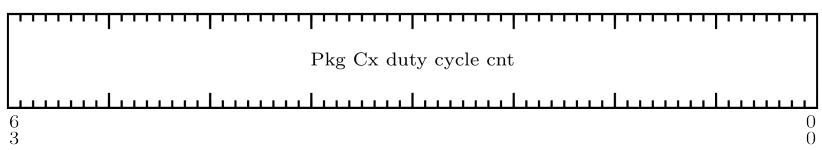
IA32_THREAD_STALL
DB2H



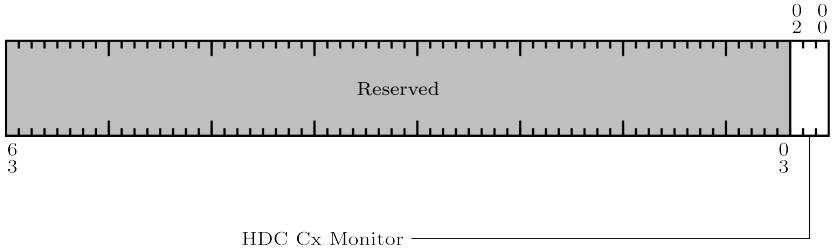
MSR_CORE_HDC_RESIDENCY
653H



MSR_PKG_HDC_SHALLOW_RESIDENCY
655H



MSR_PKG_HDC_DEEP_RESIDENCY
656H

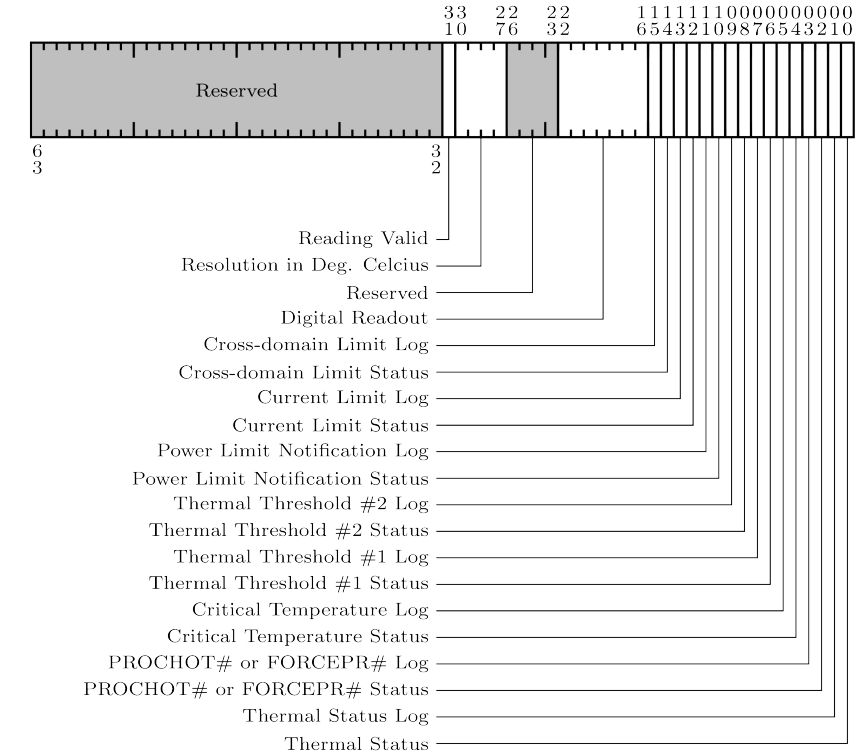


IA32_PKG_HDC_CONFIG
652H

Thermal Monitoring and Protection

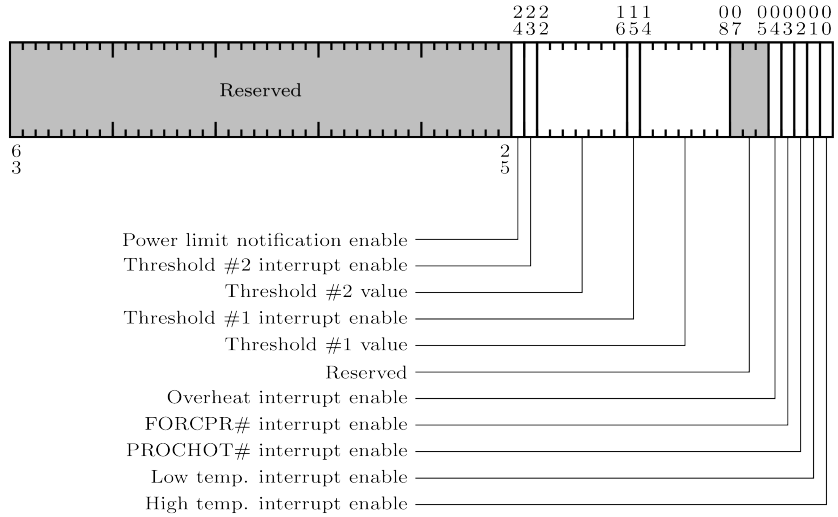
- TM1 present if CPUID.(EAX=1):EDX[29] == 1, enabled by IA32_MISC_ENABLE[3]
- TM2 present if CPUID.(EAX=1):ECX[8] == 1, enabled by IA32_MISC_ENABLE[13]
- Digital Thermal Sensor Enumeration present if CPUID.(EAX=0):EAX[0]=1
- **MSRs used**
  - MSR_THERM2_CTL
  - IA32_THERM_STATUS
  - IA32_THERM_INTERRUPT
  - IA32_CLOCK_MODULATION
  - IA32_THERM_STATUS

MSR_THERM2_CTL
NOT AVAILABLE



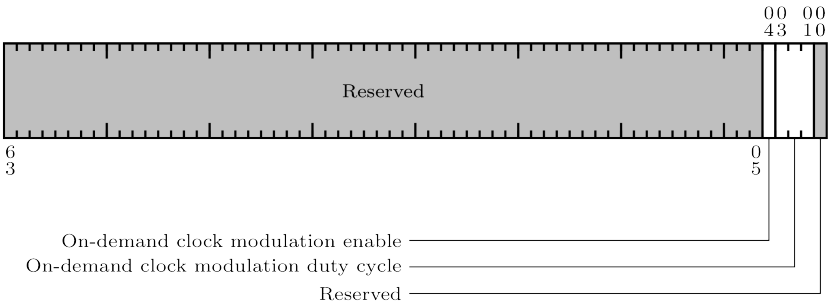
IA32_THERM_STATUS

19CH



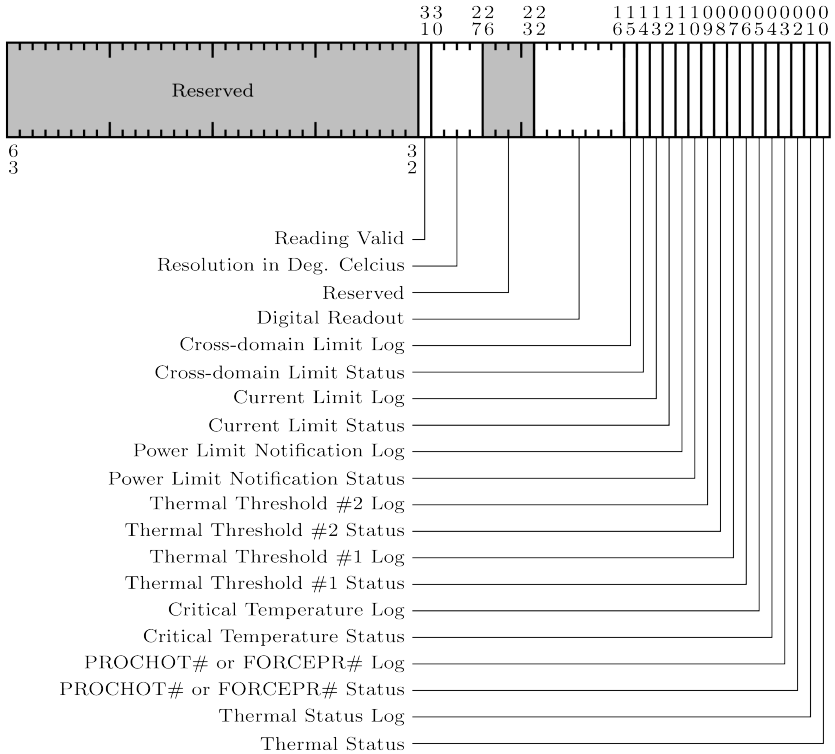
IA32_THERM_INTERRUPT

19BH



IA32_CLOCK_MODULATION

19AH

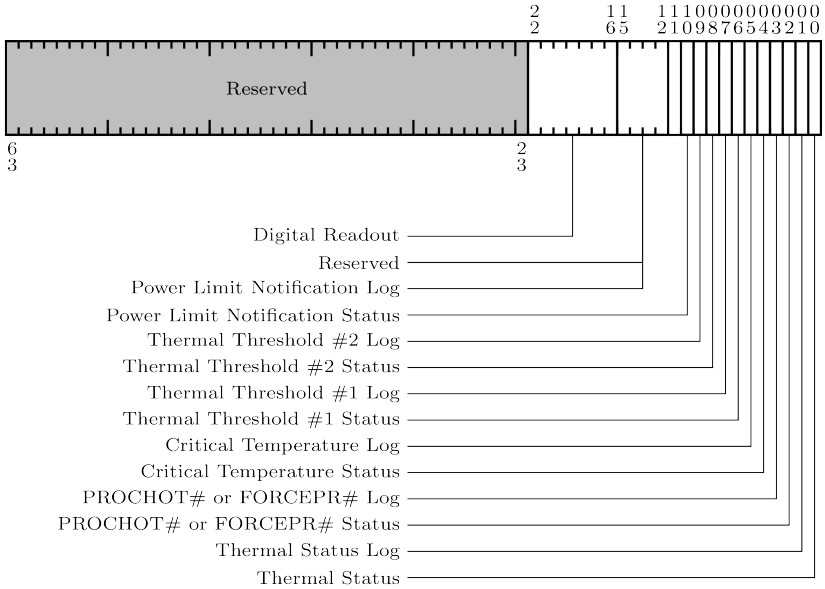


IA32_THERM_STATUS

19CH

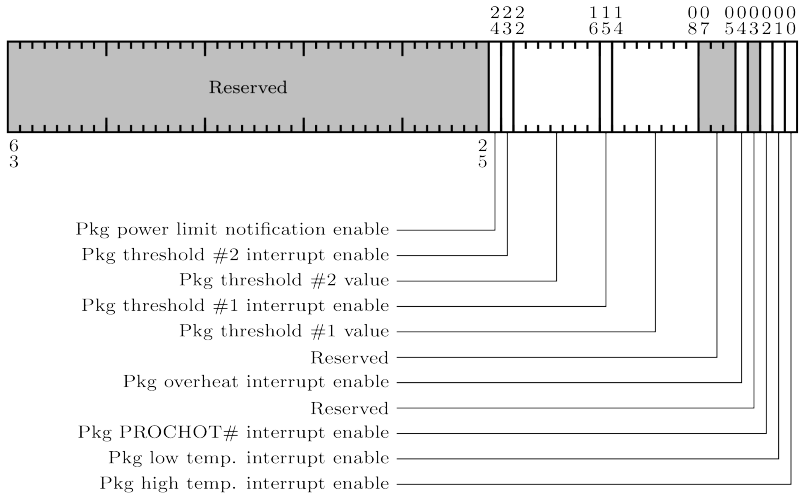
Package Level Thermal Management

- Present if CPUID.(EAX=6):EAX[6] == 1
- **MSRs used**
  - IA32_PACKAGE_THERM_STATUS
  - IA32_PACKAGE_THERM_INTERRUPT



IA32_PACKAGE_THERM_STATUS

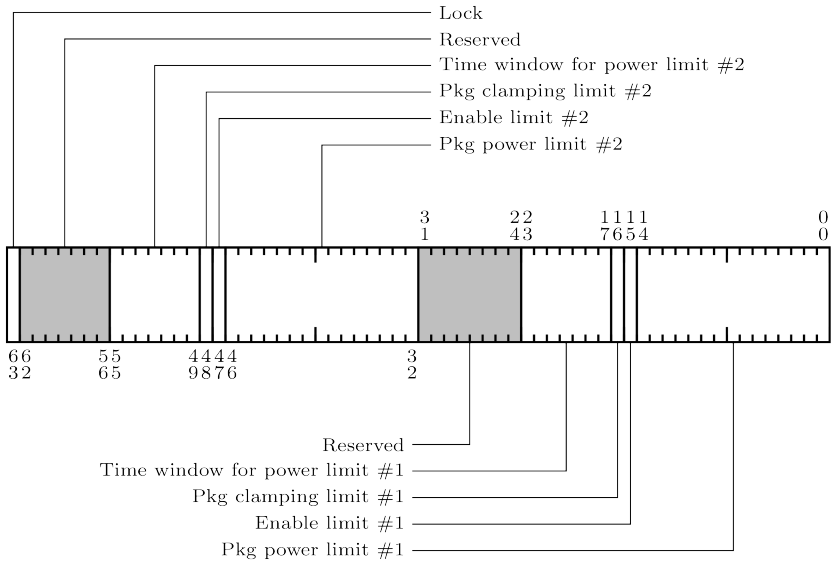
1B1H



IA32_PACKAGE_THERM_INTERRUPT
1B2H

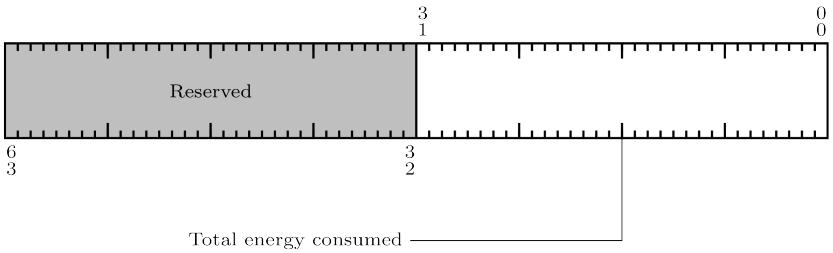
Platform Specific Power Management Support

- **MSRs used**
  - MSR_PKG_POWER_LIMIT
  - MSR_PKG_ENERGY_STATUS
  - MSR_PKG_PERF_STATUS
  - MSR_PKG_POWER_INFO
  - MSR_DRAM_POWER_LIMIT
  - MSR_DRAM_ENERGY_STATUS
  - MSR_DRAM_PERF_STATUS
  - MSR_DRAM_POWER_INFO
  - MSR_PP0_POWER_LIMIT
  - MSR_PP0_ENERGY_STATUS
  - MSR_PP0_POLICY
  - MSR_PP0_PERF_STATUS
  - MSR_PP1_POWER_LIMIT
  - MSR_PP1_ENERGY_STATUS
  - MSR_PP1_POLICY

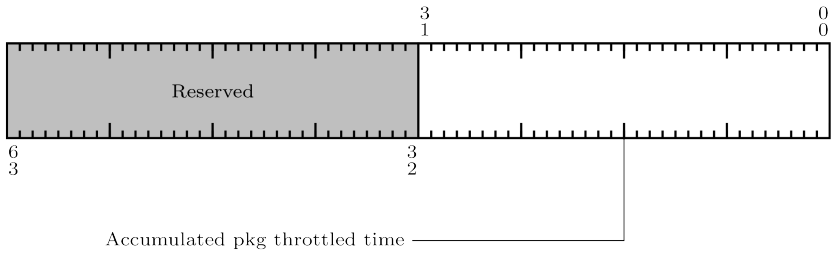


IA32_PKG_POWER_LIMIT
610H

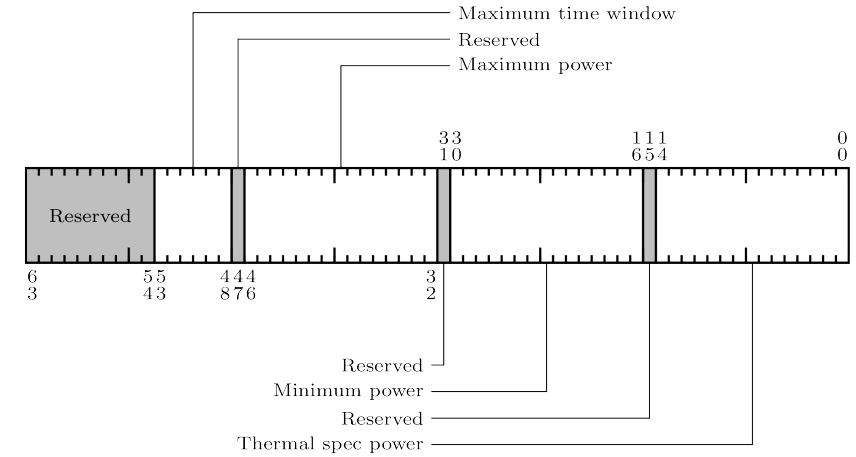
- The two different power limits use different algorithms and are intended for use across different timescales. The details are still NDA.
- There is a lower limit to the time windows. Values below that will be silently clipped. That value is also NDA.
- The OS and enable bits are now ignored. Both of them should always be set high. Writing all-zeros to this register will not disable RAPL; the processor will just try to meet a zero-watt power bound (or whatever zero is clipped to).



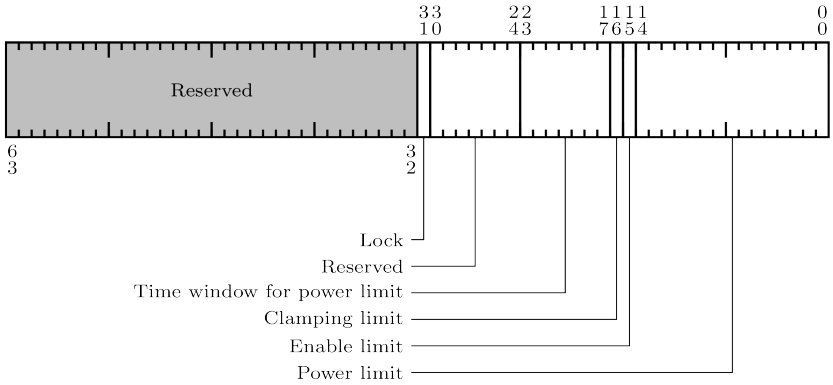
MSR_PKG_ENERGY_STATUS
611H



IA32_PKG_PERF_STATUS
613H

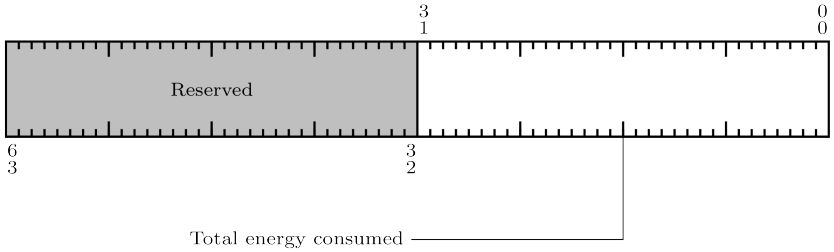


IA32_PKG_POWER_INFO
614H

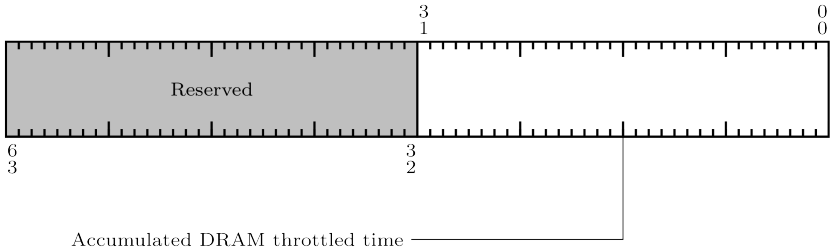


MSR_DRAM_POWER_LIMIT
618H

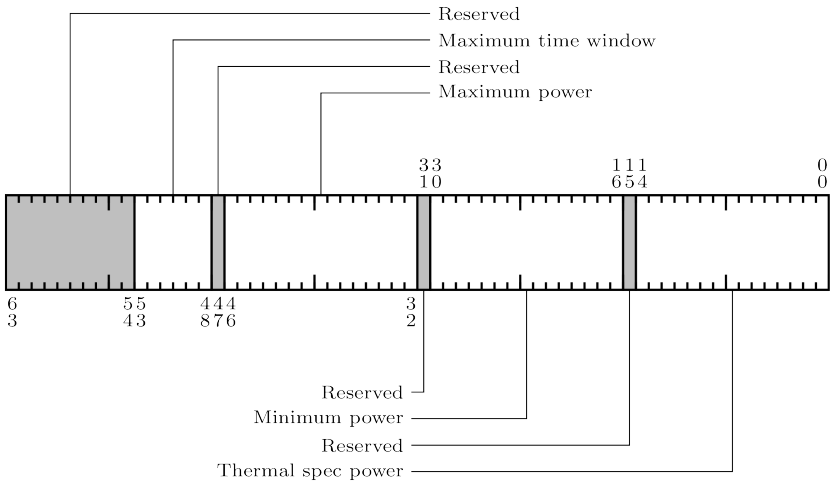
- The DRAM power controls have not proven to be that useful. If a program is not generating much memory traffic, not much power is used. Programs that do generate lots of memory traffic have outsized slowdown if memory power is restricted.



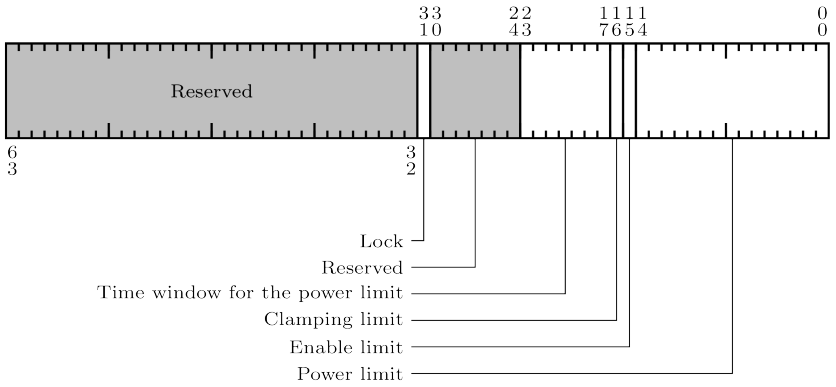
MSR_DRAM_ENERGY_STATUS
619H



MSR_DRAM_PERF_STATUS
61BH

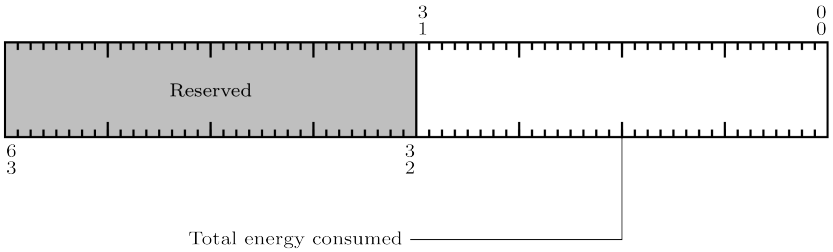


MSR_DRAM_POWER_INFO
61CH

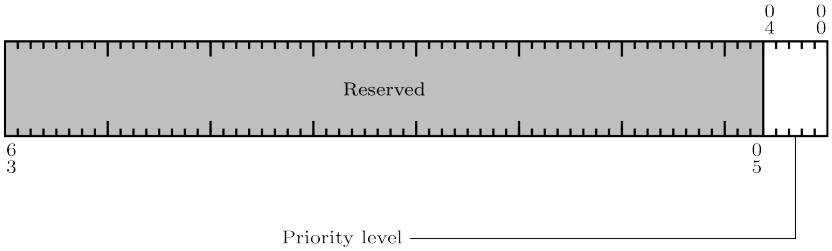


MSR_PP0_POWER_LIMIT
638H

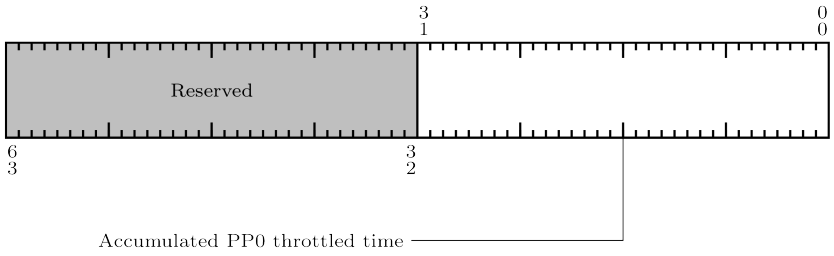
- PP0 power control has been unofficially deprecated.



MSR_PP0_ENERGY_STATUS
639H

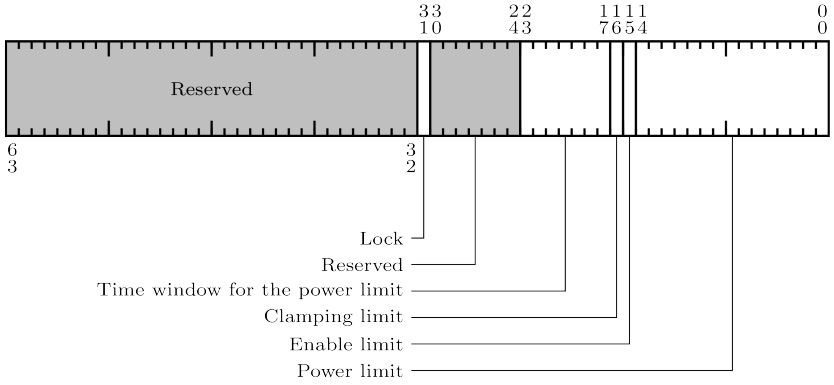


MSR_PP0_POLICY
63AH



MSR_PP0_PERF_STATUS

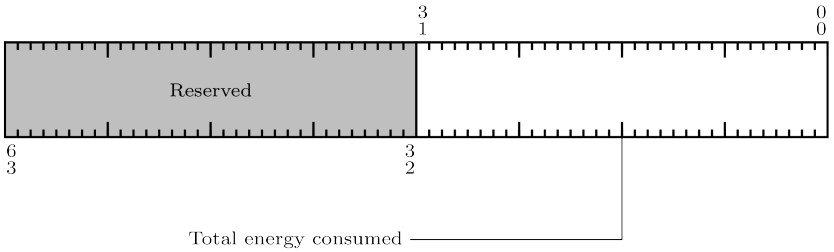
Unknown



MSR_PP1_POWER_LIMIT

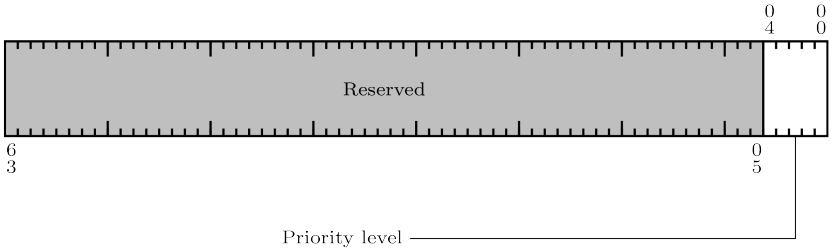
642H

- PP1 power control was intended for client processors and has not been investigated in the HPC community.



MSR_PP1_ENERGY_STATUS

641H

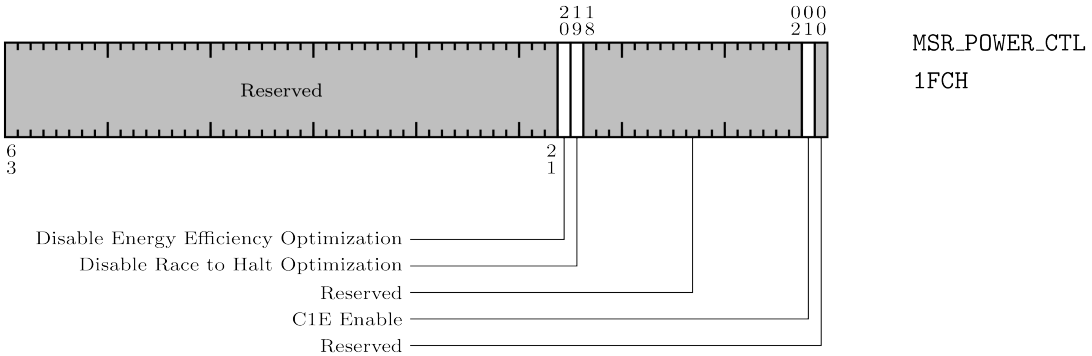


MSR_PP1_POLICY

642H

Other Public MSRs of Interest

- MSR_POWER_CTL



References

- Intel Software Developer Manuals

5.6.5 Intel Discrete GPUs Overview

This page provides a detailed description of the the Intel Discrete GPU port of Variorum. The functionality of this port depends on Intel-specific proprietary software stack as well as open-source software components described below. The high-level API provided by Variorum is currently read-only (i.e., monitoring-only), primarily because of the access limitations on our target platform.

Requirements

The Intel Discrete GPU port of Variorum depends on:

- APMIDG 0.3.0 or later
- One API 2022.06.30.002 or later

We have tested our port with Aurora early access systems with Intel ATS GPUs.

Note: at this point, monitoring power of Intel Discrete GPU requires no additional permission. However, the permission default setting may change in a future driver release. Please consult your sysadmin.

Build Configuration

At this point, Intel GPUs are only available through an early evaluation system; thus, we recommend you request your account via https://www.jlse.anl.gov/ and request access to the Aurora early access systems (EAS). Once you gain access to EAS, type the following command to load compilers and the libraries required to build the Intel GPU port.

```
$ module load cmake oneapi apmidg jansson hwloc libiconv
```

We provide an example CMake host config file, which defines the CMake build variables set on our test platform (Aurora early access supercomputer at ANL): `arcticus-apmidg-oneapi.cmake`.

If you want to build variorum on other systems besides JLSE nodes, please install all compiler and library dependencies, and then you will need to enable Variorum to build with INTEL_GPU and set three path variables as described below:

- VARIORUM_WITH_INTEL_GPU=ON
- APMDIG_DIR: Path to libapmidg.so (prefixed with the '-L' flag)
- HWLOC_DIR: Path to libhwloc.so
- JANSSON_DIR: Path to libjansson.so

Device Enumeration

The Intel GPU port enumerates the system GPU devices at initialization in the `initAPMIDG()` method, which internally obtains the number of Intel GPU devices via `apmidg_getndevs()`. The number of GPUs per socket is simply the number of available GPUs divided by the number of CPU sockets returned by `variorum_get_topology()`.

Telemetry Collection Through APMIDG Query Interface

The Intel GPU port of Variorum leverages the device and unit query APIs provided by APMIDG to collect per-GPU telemetry or subdomain telemetry if available. The text below describes the specific Variorum APIs, the corresponding APMIDG APIs, and the post-processing (if any) performed by Variorum before presenting the data to the caller.

Power telemetry

Variorum provides two APIs for power telemetry from the GPU devices:

- Average power usage

To report the average power usage of a GPU device, Variorum leverages the `apmidg_readpoweravg()` API of APMIDG. The reported power is in Watts as a floating point number.

Thermal telemetry

Variorum provides an API to report instantaneous GPU device temperature in degree Celsius and integer precision. It leverages the `apmidg_readtemp()` APMIDG API to report the GPU device temperature in Celsius.

Clocks telemetry

Variorum provides an API to report instantaneous clock speed of the Intel GPU's execusion unit in MHz and integer precision. It leverages the `apmidg_readfreq()` APMIDG API to report the instantaneous clock speed.

Control Interface

The Intel Discrete GPU port of Variorum leverages the device-level control APIs provided by APMIDG. Variorum implements the following device control APIs using the corresponding APMIDG APIs.

Power control

In Variorum's GPU power capping API, Variorum uses the `apmidg_setpwrlim()` API of APMIDG which takes as input the GPU device ID, the power domain ID and the power cap in milliwatts.

References

- APMDIG library

5.6.6 NVIDIA Overview

This page provides a detailed description of the the NVIDIA port of Variorum. The functionality of this port depends on NVIDIA-specific proprietary software stack as well as open-source software components described below. The high-level API provided by Variorum is read-only (i.e., monitoring-only), primarily because of the access limitations on our target platform.

Requirements

The NVIDIA port of Variorum depends on:

- NVIDIA Management Library (NVML) for access to the telemetry and control interfaces. NVML provides standardized interfaces to the NVIDIA GPU devices enumerated by the proprietary NVIDIA device driver as `/dev/nvidia[0-9]*`.
- CUDA development toolkit, 10.1.243+ which delivers the headers for NVML.
- CUDA-enabled build of the Portable Hardware Locality (hwloc) library to enumerate the GPU devices and their mappings to the host CPUs. This requires hwloc to be built with the `HWLOC_HAVE_CUDA` flag.

To successfully use the Variorum port of NVIDIA, verify that the `LD_LIBRARY_PATH` environment variable has paths for both the CUDA library and the CUDA-enabled hwloc library installed on the system. Also make sure that access to the NVIDIA devices (`/dev/nvidia*`) through the NVIDIA driver are set correctly for the user. This can be verified by running the *nvidia-smi* command line tool.

We have tested our NVIDIA port with CUDA 9.2 and CUDA-enabled build of hwloc 1.11.10. The NVIDIA port has been tested on the Tesla GPU architecture (NVIDIA Volta SM200).

Build Configuration

We provide an example CMake host config file, which defines the CMake build variables set on our test platform (Lassen supercomputer at LLNL): *lassen-4.14.0-ppc64le-gcc@4.9.3-cuda@10.1.243.cmake*.

For your build system, you will need to enable Variorum to build with NVIDIA and set two path variables as described below:

- `VARIORUM_WITH_NVIDIA_GPU=ON`
- `CMAKE_SHARED_LINKER_FLAGS`: Path to libnvidia-ml.so (prefixed with the '-L' flag)
- `HWLOC_DIR`: Path for the CUDA-aware version of libhwloc

Device Enumeration

The NVIDIA port enumerates the system GPU devices and populates global GPU device handles at initialization in the `initNVML()` method using the `nvmlDeviceGetCount()` and `nvmlDeviceGetHandleByIndex()` NVML query APIs, respectively. It then queries the number of CPUs using Variorum's internal routine to query system topology which uses the CUDA-enabled hwloc. Based on this information, it calculates the number of GPU devices associated with each CPU assuming sequential device assignment on the system. This method also initializes the internal state of NVML using the `nvmlInit()` API.

The device handles are stored in data structures of type `nvmlDevice_t` defined in NVML. A device handle provides the logical-to-physical mapping between the sequential device IDs and system device handles maintained by NVML internally at state initialization. All NVML query and command APIs require the device handles to perform the specified operation on the device. While the high-level Variorum APIs operate over all devices, the internal routines in the NVIDIA port use CPU ID to perform operations on the associated GPUs.

Telemetry Collection Through NVML Query Interface

The NVIDIA port of Variorum leverages the device and unit query APIs provided by NVML to collect per-GPU telemetry. The text below describes the specific Variorum APIs, the corresponding NVML APIs, and the post-processing (if any) performed by Variorum before presenting the data to the caller.

Power telemetry

Variorum provides two APIs for power telemetry from the GPU devices:

- Average power usage
- Current power limit

To report the average power usage of a GPU device, Variorum leverages the `nvmlDeviceGetPowerUsage()` API of NVML. The reported power is in Watts as an integer.

To report the power limit assigned to a GPU device, Variorum leverages the `nvmlDeviceGetPowerManagementLimit()` API of NVML. The reported power limit is in Watts as an integer.

Thermal telemetry

Variorum provides an API to report instantaneous GPU device temperature in degree Celsius and integer precision. It leverages the `nvmlDeviceGetTemperature()` NVML API to report the GPU device temperature.

Clocks telemetry

Variorum provides an API to report instantaneous Streaming Multi-processor (SM) clock speed in MHz and integer precision. It leverages the `nvmlDeviceGetClock()` NVML API to report the instantaneous SM clock speed.

Device utilization

Variorum provides an API to report the instantaneous device utilization as a percentage of time (samples) for which the GPU was in use (i.e., GPU occupancy rate) in a fixed time window. It leverages the `nvmlDeviceGetUtilizationRates()` API of NVML to report the device utilization rate as a percentage in integer precision.

Power capping

Variorum provides an API to cap GPU device power. The API applies the power cap equally to all GPU devices on the system. It leverages the `nvmlDeviceSetPowerManagementLimit()` API of NVML to set the power cap to the device after converting the specified power cap into milliwatts. This API requires root/administrator privileges.

References

- NVML API Reference

5.7 Monitoring Binaries with Variorum

While the Variorum API allows for detailed critical path analysis of the power profile of user applications as well as for integration with system software such as Kokkos, Caliper, and Flux through code annotations, there are scenarios where such annotations are not possible. In order to support such scenarios, we provide the `powmon` tool, which can monitor a binary externally with Variorum in a vendor-neutral manner. This tool can monitor an application externally without requiring any code changes or annotations.

The `variorum/src/powmon` directory contains this tool, which is built along with the regular Variorum build. While a target executable is running, `powmon` collects time samples of power usage, power limits, energy, thermals, and other performance counters for all sockets in a node at a regular interval. By default, it collects basic node-level power information, such as CPU, memory, and GPU power, at 50ms intervals, which it reports in a CSV format. It also supports a verbose (`-v`) mode, where additional registers and sensors are sampled for the advanced user. The sampling rate is configurable with the `-i` option. As an example, the command below will sample the power usage while executing a sleep for 10 seconds in a vendor neutral manner:

```
$ powmon -a "sleep 10"
```

The resulting data is written to two files:

```
hostname.powmon.dat
hostname.powmon.summary
```

Here, `hostname` will change based on the node where the monitoring is occurring. The `summary` file contains global information such as execution time. The `dat` file contains the time sampled data, such as power, thermals, and performance counters in a column-delimited format. The output differs on each platform based on available counters.

`Powmon` also supports profiling across multiple nodes with the help of resource manager commands (such as `srun` or `jsrun`) or MPI commands (such as `mpirun`). As shown in the example below, the user can specify the number of nodes through `mpirun` and utilize `powmon` with their application.

```
$ mpirun -np <num-nodes> ./powmon -a ./application
```

We also provide a set of simple plotting scripts for `powmon`, which are located in the `src/powmon/scripts` folder. The `powmon-plot.py` script can generate per-node as well as aggregated (across multiple nodes) graphs for the default

version of powmon that provides node-level and CPU, GPU and memory data. This script works across all architectures that support Variorum's JSON API for collecting power. Additionally, for IBM sensors data, which can be obtained with the powmon -v (verbose) option, we provide a post processing and R script for plots.

In addition to powmon that is vendor-neutral, for Intel systems only, we provide two other power capping tools, power_wrapper_static, and power_wrapper_dynamic that allow users to set a static (or dynamic) power cap and then monitor their binary application.

The example below will set a package-level power limit of 100W on each socket, and then sample the power usage while executing a sleep for 10 seconds:

```
$ power_wrapper_static -w 100 -a "sleep 10"
```

Similarly, the example below will set an initial package-level power limit of 100W on each socket, sample the power usage, and then dynamically adjust the power cap step-wise every 500ms while executing a sleep for 10 seconds:

```
$ power_wrapper_dynamic -w 100 -a "sleep 10"
```

5.8 Variorum Utilities

Variorum provides some utilities to assist users and developers. Currently, Variorum provides two main utilities:

- verify_opal.py: A python script to verify the system environment (i.e., OPAL) for IBM platforms.
- verify_msr_kernel.py: A python script to verify the system environment (i.e., msr kernel or msr-safe kernel) for Intel.

5.8.1 Verify OPAL

This python script verifies that the OPAL files are present and have the required R/W permissions on the target IBM hardware.

How do I use it?

From the top-level Variorum directory:

```
brink2@lassen11:~]$ python ./src/utilities/verify_opal.py -v
###################
# IBM OPAL Access #
###################
-- Check if OPAL files exist
-- Check if /sys/firmware/opal/exports/occ_inband_sensors is accessible by user: /sys/
↪firmware/opal/exports/occ_inband_sensors -- yes
-- Check if /sys/firmware/opal/powercap/system-powercap/powercap-current is accessible␣
↪by user: /sys/firmware/opal/powercap/system-powercap/powercap-current -- yes
-- Check if /sys/firmware/opal/powercap/system-powercap/powercap-max is accessible by␣
↪user: /sys/firmware/opal/powercap/system-powercap/powercap-max -- yes
-- Check if /sys/firmware/opal/powercap/system-powercap/powercap-min is accessible by␣
↪user: /sys/firmware/opal/powercap/system-powercap/powercap-min -- yes
-- Check if /sys/firmware/opal/psr/cpu_to_gpu_0 is accessible by user: /sys/firmware/
↪opal/psr/cpu_to_gpu_0 -- yes
-- Check if /sys/firmware/opal/psr/cpu_to_gpu_8 is accessible by user: /sys/firmware/
```

(continues on next page)

(continued from previous page)

```
↪opal/psr/cpu_to_gpu_8 -- yes
-- Check if OPAL files are accessible by user
-- Check if OCC file is accessible by user: /sys/firmware/opal/exports/occ_inband_
↪sensors -- yes
-- Check if powercap-current file is accessible by user: /sys/firmware/opal/powercap/
↪system-powercap/powercap-current -- yes
-- Check if powercap-max file is accessible by user: /sys/firmware/opal/powercap/system-
↪powercap/powercap-max -- yes
-- Check if powercap-min file is accessible by user: /sys/firmware/opal/powercap/system-
↪powercap/powercap-min -- yes
-- Check if cpu_to_gpu0 file is accessible by user: /sys/firmware/opal/psr/cpu_to_gpu_0 -
↪- yes
-- Check if cpu_to_gpu8 file is accessible by user: /sys/firmware/opal/psr/cpu_to_gpu_8 -
↪- yes
-- Valid OPAL access
```

Invoke the script with python on the target IBM system. The `-v` flag enables verbose output, which can be helpful if your programs are running to permissions issues. The output of this script is helpful to send to the mailing list for debugging system issues.

The last line of the output will (verbose or not) will indicate if the IBM OPAL files exist and have the appropriate permissions.

Why Do I Need This?

This is a helpful utility to run before launching any examples or tests, as it can verify that the required environment for Variorum is configured successfully.

5.8.2 Verify MSR Kernel

This utility will check if the stock msr kernel or the msr-safe kernel are loaded and configured correctly with the appropriate R/W permissions. It will first check if the msr kernel is loaded and has appropriate permissions. If this fails, then it will check if the msr-safe kernel is loaded and has appropriate permissions.

How do I use it?

From the top-level Variorum directory:

```
brink2@quartz5:~]$ python3 ./src/utilities/verify_msr_kernel.py -v
######################
# x86 CPU MSR Access #
######################
-- Check if msr_safe kernel is loaded
-- Check if msr_safe kernel is loaded -- yes
-- Check if msr_safe kernel files are character devices
-- Check if msr_safe kernel files are character devices: /dev/cpu/0/msr_safe -- yes
-- Check if msr_safe kernel files are character devices: /dev/cpu/1/msr_safe -- yes
-- Check if msr_safe kernel files are character devices: /dev/cpu/2/msr_safe -- yes
-- Check if msr_safe kernel files are character devices: /dev/cpu/3/msr_safe -- yes
...
```

(continues on next page)

(continued from previous page)

```
-- Check if msr_safe kernel files are character devices: /dev/cpu/71/msr_safe -- yes
-- Check if msr_safe kernel files are character devices: /dev/cpu/msr_allowlist -- yes
-- Check if msr_safe kernel files are character devices: /dev/cpu/msr_batch -- yes
-- Check if msr_safe kernel files are accessible by user
-- Check if msr_safe kernel files are accessible by user: /dev/cpu/0/msr_safe -- yes
-- Check if msr_safe kernel files are accessible by user: /dev/cpu/1/msr_safe -- yes
-- Check if msr_safe kernel files are accessible by user: /dev/cpu/2/msr_safe -- yes
-- Check if msr_safe kernel files are accessible by user: /dev/cpu/3/msr_safe -- yes
...
-- Check if msr_safe kernel files are accessible by user: /dev/cpu/71/msr_safe -- yes
-- Check if msr_safe kernel files are accessible by user: /dev/cpu/msr_allowlist -- yes
-- Check if msr_safe kernel files are accessible by user: /dev/cpu/msr_batch -- yes
-- Valid kernel loaded: msr-safe
```

Invoke the script with python on the target Intel system. The `-v` flag enables verbose output, which can be helpful if your programs are running to permissions issues. The output of this script is helpful to send to the mailing list for debugging system issues.

The last line of the output will (verbose or not) will indicate if the msr or msr-safe kernel is configured correctly and has the appropriate permissions.

Why Do I Need This?

This is a helpful utility to run before launching any examples or tests, as it can verify that the required environment for Variorum is configured successfully.

5.9 Variorum Print Functions

Variorum provides the following high-level functions for printing the value of various features. For each feature, there is a `print` and `print_verbose` API, which will print the metrics in different output formats. The `print` API prints the output in tabular format that can be filtered and parsed by a data analysis framework, such as `R` or `Python`. The `print_verbose` API prints the output in verbose format that is more human-readable (with units, titles, etc.). See *Examples* for sample output formats supported by Variorum.

Defined in `variorum/variorum.h`.

int **variorum_print_verbose_power**(void)

Print power usage data in long format.

Supported Architectures:

- AMD EPYC Milan
- AMD Radeon Instinct GPUs (MI50 onwards)
- ARM Juno r2
- Ampere Neoverse N1
- IBM Power9
- Intel Sandy Bridge
- Intel Ivy Bridge

- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Ice Lake
- Intel Cascade Lake
- Intel Cooper Lake
- Intel Sapphire Rapids
- NVIDIA Volta

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_power**(void)
Print power usage data in CSV format.

Supported Architectures:

- AMD EPYC Milan
- AMD Radeon Instinct GPUs (MI50 onwards)
- ARM Juno r2
- Ampere Neoverse N1
- IBM Power9
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Ice Lake
- Intel Cascade Lake
- Intel Cooper Lake
- Intel Sapphire Rapids
- Intel Arctic Sound
- NVIDIA Volta

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_verbose_power_limit**(void)

Print power limits for all known domains in long format.

Supported Architectures:

- AMD EPYC Milan
- AMD Radeon Instinct GPUs (MI50 onwards)
- IBM Power9
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Ice Lake
- Intel Cascade Lake
- Intel Cooper Lake
- Intel Sapphire Rapids
- NVIDIA Volta

Returns

0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_power_limit**(void)

Print power limits for all known domains in CSV format.

Supported Architectures:

- AMD EPYC Milan
- AMD Radeon Instinct GPUs (MI50 onwards)
- IBM Power9
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Ice Lake
- Intel Cascade Lake
- Intel Cooper Lake

- Intel Sapphire Rapids
- NVIDIA Volta

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_verbose_thermals**(void)
Print thermal data in long format.

Supported Architectures:

- AMD Radeon Instinct GPUs (MI50 onwards)
- ARM Juno r2
- Ampere Neoverse N1
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Cascade Lake
- Intel Cooper Lake
- NVIDIA Volta

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_thermals**(void)
Print thermal data in CSV format.

Supported Architectures:

- AMD Radeon Instinct GPUs (MI50 onwards)
- ARM Juno r2
- Ampere Neoverse N1
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Cascade Lake

- Intel Cooper Lake
- Intel Arctic Sound
- NVIDIA Volta

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_verbose_counters**(void)
Print performance counter data in long format.

Supported Architectures:

- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Cascade Lake
- Intel Cooper Lake

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_counters**(void)
Print performance counter data in CSV format.

Supported Architectures:

- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Cascade Lake
- Intel Cooper Lake

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_verbose_frequency**(void)

Print current operating frequency (APERF/MPERF and PERF_STATUS) in long format.

Supported Architectures:

- AMD EPYC Milan
- AMD Radeon Instinct GPUs (MI50 onwards)
- ARM Juno r2
- Ampere Neoverse N1
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Cascade Lake
- Intel Cooper Lake
- NVIDIA Volta

Returns

0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_frequency**(void)

Print current operating frequency (APERF/MPERF and PERF_STATUS) in CSV format.

Supported Architectures:

- AMD EPYC Milan
- AMD Radeon Instinct GPUs (MI50 onwards)
- ARM Juno r2
- Ampere Neoverse N1
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Cascade Lake
- Intel Cooper Lake
- Intel Arctic Sound

- NVIDIA Volta

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_hyperthreading**(void)

Print if hyperthreading is enabled or disabled.

Returns
0 if successful, otherwise -1

void **variorum_print_topology**(void)

Print architecture topology in long format.

int **variorum_print_features**(void)

Print list of features available on a particular architecture.

Supported Architectures:

- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Ice Lake
- Intel Cascade Lake
- Intel Cooper Lake
- Intel Sapphire Rapids

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_turbo**(void)

Print if turbo is enabled or disabled. If enabled, then print discrete frequencies in turbo range (i.e., max turbo ratio).

Supported Architectures:

- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_verbose_gpu_utilization**(void)

Print verbose GPU streaming multi-processor and memory utilization.

Supported Architectures:

- AMD Radeon Instinct GPUs (MI50 onwards)
- NVIDIA Volta

Architectures that do not have API support:

- AMD EPYC Milan
- IBM Power9
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Ice Lake
- Intel Cascade Lake
- Intel Cooper Lake

Returns

0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_gpu_utilization**(void)

Print CSV-formatted GPU streaming multi-processor and memory utilization.

Supported Architectures:

- AMD Radeon Instinct GPUs (MI50 onwards)
- NVIDIA Volta

Architectures that do not have API support:

- AMD EPYC Milan
- IBM Power9
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Ice Lake

- Intel Cascade Lake
- Intel Cooper Lake

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_available_frequencies**(void)

Print list of available frequencies from p-states, turbo, AVX, etc. ranges.

Supported Architectures:

- ARM Juno r2
- Ampere Neoverse N1
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Cascade Lake
- Intel Cooper Lake

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_print_energy**(void)

Print if core and socket energy is available.

Supported Architectures:

- AMD EPYC Milan

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_poll_power**(FILE *output)

Collect power limits and energy usage for both the package and DRAM domains.

Supported Architectures:

- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake

- Intel Kaby Lake
- Intel Cascade Lake
- Intel Cooper Lake

Parameters
`output` – **[in]** Location for output (stdout, stderr, filename).

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_monitoring**(FILE *output)

Collect power limits and energy usage for both the package and DRAM domains, fixed counters, TSC, APERF, and MPERF.

Supported Architectures:

- IBM Power9
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Cascade Lake
- Intel Cooper Lake
- Intel Sapphire Rapids

Parameters
`output` – **[in]** Location for output (stdout, stderr, filename).

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

char ***variorum_get_current_version**(void)

5.10 Variorum Cap Functions

Variorum provides the following high-level functions for applying an upper-bound on a particular control dial.

Defined in `variorum/variorum.h`.

int **variorum_cap_each_socket_power_limit**(int socket_power_limit)

Cap the power limits for all sockets within the node.

Supported Architectures:

- AMD EPYC Milan

- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Cascade Lake
- Intel Cooper Lake

Parameters
`socket_power_limit` – **[in]** Desired power limit for each socket.

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int `variorum_cap_best_effort_node_power_limit`(int node_power_limit)

Cap the power limit of the node.

Supported Architectures:

- IBM Power9
- AMD EPYC Milan
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Cascade Lake
- Intel Cooper Lake

Parameters
`node_power_limit` – **[in]** Desired power limit for the node.

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int `variorum_cap_gpu_power_ratio`(int gpu_power_ratio)

Cap the power shifting ratio for the GPU (uniform on both sockets).

Supported Architectures:

- IBM Power9 (same ratio on both sockets)

Architectures that do not have API support:

- AMD EPYC Milan
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Ice Lake
- Intel Cascade Lake
- Intel Cooper Lake
- NVIDIA Volta

Parameters
gpu_power_ratio – **[in]** Desired power ratio (percentage). for the processor and GPU.

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_cap_each_gpu_power_limit**(int gpu_power_limit)

Cap the power usage identically of each GPU on the node.

Supported Architectures:

- NVIDIA Volta, Ampere
- AMD Instinct (MI-50 onwards)
- Intel Discrete GPU

Parameters
gpu_power_limit – **[in]** Desired power limit in watts for each GPU on the node.

Returns
0 if successful, otherwise -1

int **variorum_cap_each_core_frequency_limit**(int cpu_freq_mhz)

Cap the CPU frequency for all cores within a socket.

Supported Architectures:

- AMD EPYC Milan
- Intel Skylake
- Intel Cascade Lake
- Intel Cooper Lake

Parameters
cpu_freq_mhz – **[in]** Desired CPU frequency for each core in MHz.

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_cap_socket_frequency_limit**(int socketid, int socket_freq_mhz)

Cap the frequency of the target processor.

Supported Architectures:

- ARM Juno r2
- Ampere Neoverse N1
- AMD EPYC Milan

Parameters

- **socketid** – **[in]** Target socket ID.
- **socket_freq_mhz** – **[in]** Desired socket frequency in MHz.

Returns
0 if successful or if feature has not been implemented or is not supported, otherwise -1

5.11 Variorum JSON-Support Functions

Variorum provides the following high-level functions that return a JSON object for easier integration with external software.

Defined in `variorum/variorum.h`.

int **variorum_get_node_power_json**(char **get_power_obj_str)

Populate a string in JSON format with total node power, socket-level, memory-level and GPU-level power.

Supported Architectures:

- AMD EPYC Milan
- ARM Juno r2
- Ampere Neoverse N1
- IBM Power9
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Cascade Lake
- Intel Cooper Lake
- Intel Sapphire Rapids

Parameters
get_power_obj_str – **[out]** String (passed by reference) that contains the node-level power information.

Returns
0 if successful, otherwise -1. Note that feature not implemented returns a -1 for the JSON APIs so that users don't have to explicitly check for NULL strings.

int **variorum_get_node_power_domain_info_json**(char **get_domain_obj_str)

Populate a string in JSON format with measurable and controllable power domains, along with the ranges.

Supported Architectures:

- AMD EPYC Milan
- ARM Juno r2
- Ampere Neoverse N1
- IBM Power9
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Cascade Lake
- Intel Cooper Lake
- Intel Sapphire Rapids

Parameters
get_domain_obj_str – **[out]** String (passed by reference) that contains the node-level domain information.

Returns
0 if successful, otherwise -1. Note that feature not implemented returns a -1 for the JSON APIs so that users don't have to explicitly check for NULL strings.

int **variorum_get_thermals_json**(char **get_thermal_obj_str)

Populate a string in nested JSON format for temperature readouts.

Format: hostname { Socket_n { CPU { Core { Sensor Name : Temp in C }, Mem { Sensor Name : Temp in C } } GPU { Device : Temp in C } } } where n is the socket number

Supported Architectures:

- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell

- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- IBM Power9
- AMD Instinct
- NVIDIA Volta

Parameters

get_thermal_obj_str – **[out]** String (passed by reference) that contains node-level thermal information.

Returns

0 if successful, otherwise -1. Note that feature not implemented returns a -1 for the JSON APIs so that users don't have to explicitly check for NULL strings.

int **variorum_get_node_frequency_json**(char **get_frequency_obj_str)

Populate a string in JSON format with node level frequency information.

Supported Architectures:

- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- IBM Power9
- AMD Instinct
- NVIDIA Volta

Parameters

get_frequency_obj_str – **[out]** String (passed by reference) that contains the node-level frequency information.

Returns

0 if successful, otherwise -1. Note that feature not implemented returns a -1 for the JSON APIs so that users don't have to explicitly check for NULL strings.

int **variorum_get_node_utilization_json**(char **get_util_obj_str)

Populate a string in JSON format with total CPU node utilization, user CPU utilization, kernel CPU utilization, total node memory utilization, and GPU utilization.

Format: { "hostname": { "CPU": { "total_util%": total_CPU_utilization, "user_util%": user_utilization, "system_util%": system_utilization, }, "GPU": { Socket_n : { GPUnm_util% : GPU_utilization }, "memory_util%": total_memory_utilization, "timestamp" : timestamp } where n is the socket number and m is the GPU id.

Supported Architectures:

- AMD Radeon Instinct GPUs (MI50 onwards)
- NVIDIA Volta

Architectures that do not have API support:

- AMD EPYC Milan
- IBM Power9
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake
- Intel Ice Lake
- Intel Cascade Lake
- Intel Cooper Lake

Parameters

`get_util_obj_str` – **[out]** String (passed by refrence) that contains node-level utilization information.

Returns

0 if successful, otherwise -1. Note that feature not implemented returns a -1 for the JSON APIs so that users don't have to explicily check for NULL strings.

int `variorum_get_gpu_utilization_json`(char **get_gpu_util_obj_str)

Populate a string in JSON format with utilization of each GPU.

Format: { "hostname": { "GPU": { Socket_n : { GPUnm_util% : GPU_utilization }, "timestamp" : timestamp } where n is the socket number and m is the GPU ID.

Supported Architectures:

- AMD Radeon Instinct GPUs (MI50 onwards)
- NVIDIA Volta

Architectures that do not have API support:

- AMD EPYC Milan
- IBM Power9
- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell
- Intel Skylake
- Intel Kaby Lake

- Intel Ice Lake
- Intel Cascade Lake
- Intel Cooper Lake

Parameters

get_gpu_util_obj_str – **[out]** String (passed by refrence) that contains node-level utilization information.

Returns

0 if successful, otherwise -1. Note that feature not implemented returns a -1 for the JSON APIs so that users don't have to explicily check for NULL strings.

5.12 Variorum Enable/Disable Functions

Variorum provides the following high-level functions for toggling a control dial.

Defined in `variorum/variorum.h`.

int **variorum_enable_turbo**(void)

Enable turbo feature.

Supported Architectures:

- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell

Returns

0 if successful or if feature has not been implemented or is not supported, otherwise -1

int **variorum_disable_turbo**(void)

Disable turbo feature.

Supported Architectures:

- Intel Sandy Bridge
- Intel Ivy Bridge
- Intel Haswell
- Intel Broadwell

Returns

0 if successful or if feature has not been implemented or is not supported, otherwise -1

5.13 Variorum Topology Functions

As part of Variorum's advanced API for developers and contributors, we provide a `hwloc` topology structure and some basic functionality for our users. These functions are described below.

Defined in `variorum/variorum_topology.h`.

int **variorum_get_num_sockets**(void)

Get number of sockets on the hardware platform.

Returns
Number of sockets, otherwise -1

int **variorum_get_num_cores**(void)

Get number of cores on the hardware platform.

Returns
Number of cores, otherwise -1

int **variorum_get_num_threads**(void)

Get number of threads on the hardware platform.

Returns
Number of threads, otherwise -1

5.14 JSON API

5.14.1 Obtaining Thermal Information

The API to obtain node thermal has the following format. It takes a string (`char**`) by reference as input, and populates this string with a nested JSON object with hostname, followed by socket_{number}, followed by CPU and or GPU (depending on the platform, may contain only one or both), followed by Core and Mem for CPU.

The `variorum_get_thermals_json(char **)` function returns a string type nested JSON object. An example is provided below:

```
{
    "hostname": {
        "Socket_0": {
            "CPU":  {
                "Core": {
                    "temp_celsius_core_0":  (Integer),
                     ...
                    "temp_celsius_core_i":  (Integer),
                },
                "Mem":  {
                    "temp_celsius_dimm_0":  (Integer),
                     ...
                    :temp_celsius_dimm_i":  (Integer),
                 },
            },
            "GPU":  {
                "temp_celsius_gpu_0":  (Integer),
                 ...
```

(continues on next page)

(continued from previous page)

```
                "temp_celsius_gpu_i":  (Integer),
            }
        },
        "timestamp" : (Integer)
    }
}
```

Here, `i` is the index of the core or GPU and `0 <= i < num_cores/GPUs`.

5.15 Contributing Guide

This page is for those who would like to contribute a new feature or bugfix to Variorum. The guide will show a few examples of contributing workflows and discuss the granularity of pull requests (PRs). It will also discuss the tests your PR must pass in order to be accepted into Variorum.

The changes proposed in a PR should correspond to one feature, bugfix, etc. PRs can contain changes relevant to different ideas, however reviewing these PRs becomes tedious and error prone. If possible, try to follow the **one-PR-per-bugfix/feature** rule.

5.15.1 Branches

Variorum is in active development. The `dev` branch has the latest contributions to Variorum. All pull requests should start from the `dev` branch and target `dev`.

There is a branch for each release, each originating from `dev`, and follows the naming convention `releases/v<major>.<minor>.x`.

5.15.2 Continuous Integration

Variorum uses Github Actions for Continuous Integration testing. For each pull request, a series of tests will be run to make sure the code change does not accidentally introduce any bugs into Variorum. Your PR must pass all of these tests before being accepted. While you can certainly wait for the results of these tests after submitting a PR, we recommend that you run them locally to speed up the review process.

We currently test against several gcc versions and different build options on Linux and perform 2 types of tests:

Unit Tests

Unit tests ensure that core Variorum features like building for supported platforms are working as expected.

The status of the unit tests can be checked on the Github Actions tab.

These tests may take several minutes to complete depending on how quickly the runners are launched.

Style Tests

Variorum uses two formatters for style checking. Flake8 to test for PEP 8 conformance. PEP 8 is a series of style guides for Python that provide suggestions for everything from variable naming to indentation. Flake8 is a code linter; it provides warnings of syntax errors, possible bugs, stylistic errors, etc.

To check for compliance, you will want to run the following two commands at the root of Variorum:

```
$ flake8
$ black --check --diff --exclude "/(src/thirdparty_builtin/googletest|build|src/docs)/" .
```

If your code is compliant, `flake8` will output nothing:

```
$ flake8
```

and `black` will show:

```
$ black --check --diff --exclude "/(src/thirdparty_builtin/googletest|build|src/docs)/" .

All done!
4 files would be left unchanged.
```

However, if your code is not compliant with PEP 8, flake8 and black will complain:

```
$ flake8
./src/utilities/verify_msr_kernel.py:35:1: E302 expected 2 blank lines, found 1
./src/utilities/verify_opal.py:12:1: F401 'numpy' imported but unused
```

```
$ black --check --diff --exclude "/(src/thirdparty_builtin/googletest|build|src/docs)/" .
--- src/utilities/verify_opal.py     2022-07-07 05:09:42.145667 +0000
+++ src/utilities/verify_opal.py     2022-07-07 05:09:46.232596 +0000
@@ -10,10 +10,11 @@
 import os
 import sys
 import pandas

 import getopt
+

 def check_opal_files_presence(verbose):
     if verbose:
         print("-- Check if OPAL files exist")

would reformat src/utilities/verify_opal.py
Oh no!
1 file would be reformatted, 3 files would be left unchanged.
```

As you address these errors with the addition or removal of lines, the line numbers will change, so you will want to re-run flake8 and black again to update them.

Alternatively, fixing the errors in reverse order will eliminate the need for multiple runs of flake8 and black just to re-compute line numbers.

Additionally, Variorum uses Artistic Style for formatting C/C++ files.

Note: We have a helper script in Variorum for calling `astyle` locally and checking for style compliance of your C/C++ files. To call this script to format C/C++ files, use `scripts/check-code-format.sh`.

5.15.3 Contributing Workflow

(Thanks to Spack for providing a great overview of the different contributing workflows described in this section.)

Variorum is under active development, so new features and bugfixes are constantly being merged into the `dev` branch. The recommended way to contribute a pull request is to fork the Variorum repository in your own space (if you already have a fork, make sure is it up-to-date), and then create a new branch off of `dev`.

```
$ git checkout dev
$ git fetch upstream && git merge --ff-only upstream/dev
$ git branch <descriptive_branch_name>
$ git checkout <descriptive_branch_name>
```

Here, we assume that the upstream remote points at `https://github.com/llnl/variorum.git`.

We prefer that commits pertaining to different pieces of Variorum (new hardware port, specific hardware feature, docs, etc.) prefix the component name in the commit message (for example `<component>: descriptive message`.

Now, you can make your changes while keeping the `dev` branch unmodified. Edit a few files and commit them by running:

```
$ git add <files_to_be_part_of_the_commit>
$ git commit --message <descriptive_message_of_this_particular_commit>
```

Next, push it to your remote fork (that is, `origin` points at `https://github.com/<your_user_name>/variorum.git`) and create a PR:

```
$ git push origin <descriptive_branch_name>
```

GitHub provides a tutorial on how to file a pull request. When you send the request, make `dev` the destination branch.

If you have multiple PRs that build on top of one another, one option is to keep a branch that includes all of your other feature branches:

```
$ git checkout dev
$ git branch <your_branch_with_all_features>
$ git checkout <your_branch_with_all_features>
$ git rebase <descriptive_branch_name>
```

This can be done with each new PR you submit. Just make sure to keep this local branch up-to-date with upstream `dev` too.

Rebasing

Other developers are making contributions to Variorum, possibly to the same files that your PR has modified. If their PR is merged before yours, it can create a merge conflict. This means that your PR can no longer be automatically merged without a chance of breaking your changes. In this case, you will be asked to rebase your branch on top of the latest upstream `dev`.

First, make sure your `dev` branch is up-to-date:

```
$ git checkout dev
$ git fetch upstream
$ git merge --ff-only upstream/dev
```

Now, we need to switch to the branch you submitted for your PR and rebase it on top of `dev`:

```
$ git checkout <descriptive_branch_name>
$ git rebase dev
```

Git will likely ask you to resolve conflicts. Edit the file that it says can't be merged automatically and resolve the conflict. Then, run:

```
$ git add <file_with_a_conflict>
$ git rebase --continue
```

You may have to repeat this process multiple times until all conflicts are resolved. Once this is done, simply force push your rebased branch to your remote fork:

```
$ git push --force origin <descriptive_branch_name>
```

5.16 Variorum Developer Documentation

Here is some information on how to extend Variorum to support additional platforms and/or microarchitectures.

A **platform** refers to a hardware vendor, for example, Intel, IBM, or ARM. A **microarchitecture** refers to a generation of hardware within a platform, for example, Broadwell or Ivy Bridge for Intel Xeon processors; or Power8 and Power9 for IBM Power processors.

5.16.1 Steps to Add Support for a New Platform (pfm)

1. Create a new folder under `src/variorum/pfm`
2. Create a `config_pfm.h` and `config_pfm.c`. These two files implement the `detect_pfm_arch` and `set_pfm_func_ptrs` functions. The former identifies the new platform, and the latter sets function pointers for get/set power, thermals, etc. on the new platform. These function pointers refer to internal functions that are defined in microarchitecture-specific files (see next section).
3. Add a struct listing all the microarchitectures implemented on the new platform in `src/variorum/config_architecture.c`. Refer to the enum for Intel microarchitectures as an example. You may need to check for the platform in a few places in the `config_architecture.c` file. Examples of Intel and IBM are included, so these can be referred to.
4. If you need to modify front-facing APIs, add them to `variorum.h` and `variorum.c`.

5.16.2 Steps to Add Support for a New Microarchitecture (pfm_arch)

1. Follow the steps listed above to create a new platform if the platform does not already exist in the Variorum source.
2. For each microarchitecture, add a `pfm_arch.h` and `pfm_arch.c` file, define the internal get/set functions for capturing power, thermal, performance data. These need to be added as function pointers in the platform file (`config_pfm.h` and `config_pfm.c` files).
3. The internal implementation will depend on the interfaces, such as sensors, MSRs, OPAL, IPMI, etc. If applicable, these can be re-used across microarchitectures (i.e., the same implementation is used for many microarchitectures).

5.16.3 Example

As an example, to support additional NVIDIA microarchitectures: 1. Under the `Nvidia/` directory, create a `.h` and `.c` header and source file for the respective microarchitecture. This will contain features specific to that microarchitecture, which may or may not exist in previous generations.

2. Modify `Nvidia/config_nvidia.c` to set the function pointers for the respective microarchitecture.

3. Include the new header file in `Nvidia/config_architecture.h`.

5.17 Variorum Unit Tests

Variorum code is regularly tested on a diverse set of architectures, including several Intel platforms (such as Broadwell, Haswell, Skylake, Ice Lake), IBM Power 9, ARM Juno r2, NVIDIA Volta GPUs, and AMD Instinct GPUs. Variorum's unit tests are run externally on GitHub, as well as internally on our Livermore Computing clusters through GitLab.

For Variorum pull requests, we use GitHub Actions for CI testing, the worflow can be found here: Variorum GitHub Actions.

Additionally, we utilize GitLab runners on our internal clusters for further CI testing on known hardware architectures, such as Quartz, Lassen, and Corona (see clusters at LLNL).

Within one of our GitLab runners, we are also leveraging Ansible to expand our CI testing across an additional set of hardware architectures. These systems require specific requests and permissions to gain access.

As part of Variorum's CI testing, we cover the following:

- Verifying compliance of code format for C/C++, RST, and Python files
- Building Variorum with different versions of `gcc` compiler
- Building Variorum with different CMake build options
- Building Variorum using Spack mirrors
- Running Variorum's unit tests and examples (for example, `make test` and `variorum-print-power-example`)

5.18 Integrating with Variorum

Variorum is a node-level library that can be integrated easily with higher-level system software such as schedulers and runtime systems, to create a portable HPC PowerStack. As part of our efforts to support a hierarchical, dynamic and open-source portable power management stack, we have integrated Variorum with various open-source system software. The JSON API enables Variorum to interface with higher-level system software in an portable and easy manner.

5.18.1 ECP Integrations

Current integration efforts include a Kokkos connector for power monitoring, a Caliper service for method-level power data, a Flux power management module for scheduling, and Sandia's OVIS Lightweight Distributed Metric Service (LDMS) monitoring plugin. These enable Variorum to be used at scale across multiple layers of the PowerStack, application/user layer, resource management layer, and system-level layer. Upcoming integrations also include developing a Variorum interface for PowerAPI and Intel GEOPM.

Links to Variorum's integrations with each of these frameworks can be found below. Note that these integrations are in early development stages and are expected to be updated to support more features and tests.

- Variorum Kokkos connector
- Variorum Caliper service
- Flux System Power Manager Module with Variorum
- LDMS Power Monitoring Plugin with Variorum

5.18.2 Contributing Integrations with JSON

In order for tools to interact with Variorum, a simple JANSSON based parser is sufficient. Our existing integration implementations, which are linked here, are a good starting point.

The format of the JSON object has been documented in *Variorum API* and includes total node power, as well as CPU, Memory, and GPU power (current assumption is that of two sockets per node). It also includes the hostname and the timestamp.

We describe a simple example of how the data from the JSON object can be extracted from Variorum in the client tool below, and we point developers to the Kokkos and Flux integration links shown above for more detailed examples. We have used the JANSSON library for our purposes, but the JSON object can also be retrieved in a similar manner by other JSON libraries and supporting tools.

```
#include <jansson.h>

void parse_variorum_data()
{
    // Define a JSON object to retrieve data from Variorum in
    json_t *power_obj = json_object();
    char *s = NULL;

    // Define a local variable for the value of interest. For example, the
    // client side tool may only be interested in node power, or it may be
    // collecting samples for integration.
    double power_node;
```

(continues on next page)

(continued from previous page)

```
    // Define a variable to capture appropriate return code from Variorum
    int ret;

    // Call the Variorum JSON API
    ret = variorum_get_node_power_json(&s);
    if (ret != 0)
    {
        printf("Variorum get node power API failed.\n");
        free(s);
        exit(-1);
    }

    // Extract the value of interest from the JSON object by using the
    // appropriate get function. Documentation of these can be found in the
    // JANSSON library documentation.
    power_obj = json_loads(s, JSON_DECODE_ANY, NULL);
    power_node = json_real_value(json_object_get(power_obj, "power_node_watts"));
    printf("Node power is: %lf\n", power_node);

    // Decrement references to JSON object, required for JANSSON library.
    json_decref(power_obj);

     // Deallocate the string
    free(s);
}
```

5.19 ECP Argo Project

Argo is a collaborative project between Argonne National Laboratory and Lawrence Livermore National Laboratory. It is funded by the U.S. Department of Energy as part of the Exascale Computing Project (ECP).The goal of the Argo project is to augment and optimize existing Operating Systems/Runtime components for use in production HPC systems, providing portable, open source, integrated software that improves the performance and scalability of and that offers increased functionality to exascale applications and runtime systems. Argo software is developed as a toolbox – a collection of autonomous components that can be freely mixed and matched to best meet the user's needs. The project has four focus areas:

- Memory management
- Power management
- Resource management
- Hardware co-design

Variorum is a key player for the *power management* and *hardware co-design* focus areas for Argo. Details about the broader Argo project, along with relevant source code, publications, and accomplishments can be found on the Argo website. The Argo project is also a key contributor to the *HPC PowerStack Initiative*, which is a community effort for power management.

Dynamic power management is expected to be critical in the exascale era, both in terms of not exceeding the overall available power budget and in terms of utilizing the available power to make the most application progress. Argo employs hierarchical power management that includes both system-level (global) and node-local mechanisms and policies. This includes developing and enabling power management in HPC schedulers (such as SLURM and Flux), large-scale

system monitoring frameworks (LDMS), application profilers (such as Caliper), runtime system frameworks (such as Kokkos, GEOPM), and similar system- and application-level frameworks.

Hardware co-design involves working closely with our vendors to continuously explore emerging new hardware trends and devices, and look for how best to exploit the new capabilities in both traditional HPC workloads and emerging scientific workflows, ones such as machine learning. We try to identify how the new features should be integrated into existing runtime systems and operating systems for the exascale era.

5.20 HPC PowerStack Initiative

The HPC PowerStack Initiative brings together experts from academia, research laboratories and industry in order to design a holistic and extensible power management framework, which we refer to as the PowerStack. The PowerStack explores hierarchical interfaces for power management at three specific levels: batch job schedulers, job-level runtime systems, and node-level managers. The HPC PowerStack community meets regularly, details of which can be found at their website.

The *ECP Argo Project* is one of the key contributors to the HPC PowerStack Initiative. HPC PowerStack also closely collaborates with the following community efforts:

5.20.1 PowerAPI

The Power API is a community-lead specification for an power monitoring and control API that supports small scale to extreme-scale systems. It is associated with the Energy-Efficient High Performance Computing Working Group (EEHPCWG) and also with the Exascale Computing Project (ECP). A community specification as well as a reference implementation of the Power API can be found on their website, https://pwrapi.github.io.

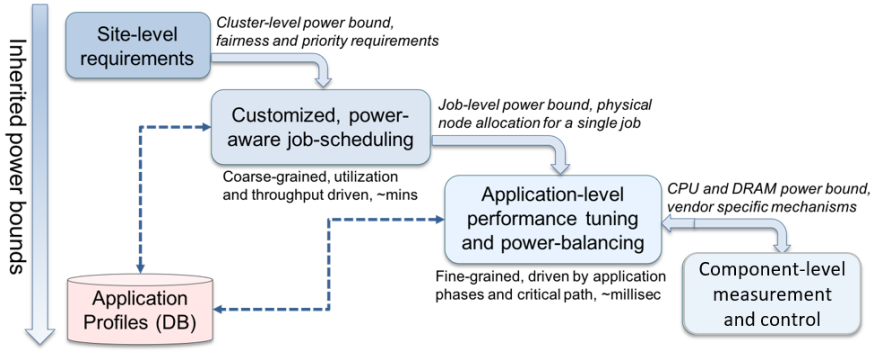
Upcoming goal for the Variorum team includes supporting relevant platform- and node-level APIs from the Power API community specification.

5.20.2 EEHPC WG

The Energy Efficient High Performance Computing Working Group (EEHPC WG) is a broad community effort focused on sustainably supporting science through committed community action. It has 800+ members worldwide, which include 50% sites, 30% industry, and 20% academia. With the help of 15+ teams addressing various opportunities for improving energy efficiency, the EEHPC WG facilitates engagement of the community in sustainable large-scale science. They also regularly organize workshops and BoFs at major HPC conferences. More details about the EEHPC WG can be found at: https://eehpcwg.llnl.gov/.

Each level in the PowerStack will provide options for adaptive and dynamic power management depending on requirements of the supercomputing site under consideration, and would be able to operate as an independent entity if needed. Site-specific requirements such as cluster-level power bounds, user fairness, or job priorities will be translated as inputs to the job scheduler. The job scheduler will choose power-aware scheduling plugins to ensure compliance, with the primary responsibility being management of allocations across multiple users and diverse workloads. Such allocations (physical nodes and job-level power bounds) will serve as inputs to a fine-grained, job-level runtime system to manage specific application ranks, in-turn relying on vendor-agnostic node-level measurement and control mechanisms.

The figure below presents an overview of the envisioned PowerStack, which takes a holistic approach to power management. Variorum is a reference example that represents the vendor-agnostic node-level interface, which can integrate with high-levels in the PowerStack as and when needed, which we describe in *Integrating with Variorum*.



5.21 Publications and Presentations

5.21.1 Publications

5.21.2 Presentations

- Variorum: Vendor-Agnostic Power Management Watch here
- Power Management on Exascale Platforms with Variorum, ECP Tutorial Series 2023.
  - Introduction to Variorum Recording
  - Integrating Variorum with System Software and Tools Recording
- Introduction to Variorum, ECP Tutorial Series 2021.
  - `Module 1 Slides` | Module 1 Recording
  - `Module 2 Slides` | Module 2 Recording
- Managing Power Efficiency of HPC Applications with Variorum and GEOPM, ECP 2020 Tutorial, February 4, 2020, Houston, TX.
  - `Part I Slides`
  - `Part II Slides`
- HPC PowerStack: Community-driven Collaboration on Power-aware System Stack, SC19 BOF, November 20, 2019, Denver, CO.
- PowerStack Seminar, November 13-15, 2019, Colorado Springs, CO.
- PowerStack Seminar, June 12-14, 2019, Garching, Germany.
- Boosting Power Efficiency of HPC Applications with GEOPM, ISC Tutorial, June 16, 2019, Frankfurt, Germany.

5.22 Contributors

We would like to acknowledge previous and current contributors to the variorum project (in alphabetical order along with their current affiliation). These include our academic, industrial and DOE collaborators without whose support this work would not be possible. We thank them for the regular discussions, bug fixes, feature enhancements, testbed access and other support.

- Mohammad Al-Tahat (New Mexico State University)
- Peter Bailey (Google)
- Natalie Bates (EEHPCWG)
- Shilpasri Bhat (IBM)
- Sascha Bischoff (ARM)
- Grayson Blanks (Apple)
- Chaz-Akil Browne (Oakwood University)
- Naveen Krishna Chatradhi (AMD)
- Jeff Booher-Kaeding (ARM)
- Stephanie Brink (LLNL)
- Christopher Cantalupo (Intel)
- Giridhar Chukkapalli (NVIDIA)
- Tiffany Connors (NERSC)
- Rigoberto Delgado (LLNL)
- Jonathan Eastep (Intel)
- Daniel Ellsworth (Colorado College)
- Dave Fox (LLNL)
- Neha Gholkar (Intel)
- Ryan Grant (Queens University, Canada)
- Eric Green (LLNL)
- Jessica Hannebert (Colorado College)
- Sachin Idgunji (NVIDIA)
- Kamil Iskra (Argonne National Laboratory)
- Tanzima Islam (Texas Tech University)
- Siddhartha Jana (EEHPCWG/Intel)
- Masaaki Kondo (University of Tokyo)
- Naman Kulshreshtha (Clemson University)
- Eun K Lee (IBM)
- David Lowenthal (Univeristy of Arizona)
- Aniruddha Marathe (LLNL)
- Matthias Maiterth (TU Munich, Germany)

- Lauren Morita (LLNL)
- Frank Mueller (North Carolina State University)
- Swann Perarnau (Argonne National Laboratory)
- Tapasya Patki (LLNL)
- Todd Rosedahl (IBM)
- Barry Rountree (LLNL)
- Roxana Rusitoru (ARM)
- Ryuichi Sakamoto (University of Tokyo, Japan)
- Siva Sathappan (AMD)
- Martin Schulz (TU Munich, Germany)
- Kathleen Shoga (LLNL)
- Carsten Trinitis (TU Munich, Germany)
- Scott Walker (Intel)
- Torsten Wilde (HPE)
- Kazutomo Yoshii (Argonne National Laboratory)

5.23 Contributor Covenant Code of Conduct

5.23.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

5.23.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

5.23.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

5.23.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail

5.24 Releases

Variorum is under constant development. So, we recommend using our `dev` branch, which contains our latest features.

5.24.1 v0.7.0

06/12/2023: Major release adds support for new architectures, ARM Neoverse N1 platform and Sapphire Rapids, enables heterogeneous build support for two architectures, adds power capping API for NVIDIA, AMD and Intel GPUs, adds get power limit for Intel GPUs, overhauls powmon utility, updates filenames and low-level function names to include architecture, updates logo. v0.7.0 tarball here.

5.24.2 v0.6.0

09/15/2022: Major release adds support for two new architectures, Intel discrete GPUs and AMD GPUs, adds python wrappers and a pyVariorum python module, updates existing JSON APIs to use char* instead of json_t*, creates new variorum_topology.h header to expose get topology APIs, moves VARIORUM_LOG from a build variable to an environment variable, standardizes examples with -h and -v flags. v0.6.0 tarball here.

5.24.3 v0.5.0

06/22/2022: Major release adds support for new architectures, AMD CPUs and Intel Ice Lake CPU, adds examples for integrating Variorum into OpenMP and MPI programs, adds dependency on rankstr library for MPI examples (optional), renames clock_speed to frequency and power_limits to power_limit throughout Variorum API, deprecates set_and_verify API. v0.5.0 tarball here.

5.24.4 v0.4.1

04/02/2021: New release adds documentation for Nvidia port, adds units to the tabular and JSON output formats, and finishes renaming dump and print internal functions to print and print_verbose, respectively. v0.4.1 tarball here.

5.24.5 v0.4.0

03/03/2021: Updated version includes support for ARM Juno architecture, introduction of the JSON API, and Intel support for two versions of msr-safe. v0.4.0 tarball here.

5.24.6 v0.3.0

04/23/2020: Updated version includes general infrastructure updates, such as unit tests, example integrations, and small bug fixes throughout. v0.3.0 tarball here.

5.24.7 v0.2.0

03/13/2020: Updated version includes support for Nvidia GPGPUs. v0.2.0 tarball here.

5.24.8 v0.1.0

11/11/2019: Initial release of Variorum. v0.1.0 tarball here.

5.25 License Info

5.25.1 Variorum Notice

5.25.2 Variorum License

5.25.3 Third Party Builtin Libraries

Here is a list of the software components used by Variorum in source form and the location of their respective license files in our source repo.

C and C++ Libraries

- *gtest*: thirdparty_builtin/gtest-1.7.0/LICENSE (BSD Style License)

Build System

- *CMake*: http://www.cmake.org/licensing/ (BSD Style License)
- *Spack*: http://software.llnl.gov/spack (MIT/Apache License)

Documentation

- *sphinx*: http://sphinx-doc.org/ (BSD Style License)
- *breathe*: https://github.com/michaeljones/breathe (BSD Style License)
- *rtd sphinx theme*: https://github.com/snide/sphinx_rtd_theme/blob/main/LICENSE (MIT License)

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

V

variorum_cap_best_effort_node_power_limit (*C++ function*), 64
variorum_cap_each_core_frequency_limit (*C++ function*), 65
variorum_cap_each_gpu_power_limit (*C++ function*), 65
variorum_cap_each_socket_power_limit (*C++ function*), 63
variorum_cap_gpu_power_ratio (*C++ function*), 64
variorum_cap_socket_frequency_limit (*C++ function*), 66
variorum_disable_turbo (*C++ function*), 70
variorum_enable_turbo (*C++ function*), 70
variorum_get_current_version (*C++ function*), 63
variorum_get_gpu_utilization_json (*C++ function*), 69
variorum_get_node_frequency_json (*C++ function*), 68
variorum_get_node_power_domain_info_json (*C++ function*), 67
variorum_get_node_power_json (*C++ function*), 66
variorum_get_node_utilization_json (*C++ function*), 68
variorum_get_num_cores (*C++ function*), 71
variorum_get_num_sockets (*C++ function*), 71
variorum_get_num_threads (*C++ function*), 71
variorum_get_thermals_json (*C++ function*), 67
variorum_monitoring (*C++ function*), 63
variorum_poll_power (*C++ function*), 62
variorum_print_available_frequencies (*C++ function*), 62
variorum_print_counters (*C++ function*), 58
variorum_print_energy (*C++ function*), 62
variorum_print_features (*C++ function*), 60
variorum_print_frequency (*C++ function*), 59
variorum_print_gpu_utilization (*C++ function*), 61
variorum_print_hyperthreading (*C++ function*), 60
variorum_print_power (*C++ function*), 55
variorum_print_power_limit (*C++ function*), 56
variorum_print_thermals (*C++ function*), 57
variorum_print_topology (*C++ function*), 60
variorum_print_turbo (*C++ function*), 60
variorum_print_verbose_counters (*C++ function*), 58
variorum_print_verbose_frequency (*C++ function*), 58
variorum_print_verbose_gpu_utilization (*C++ function*), 60
variorum_print_verbose_power (*C++ function*), 54
variorum_print_verbose_power_limit (*C++ function*), 55
variorum_print_verbose_thermals (*C++ function*), 57