GEOPM Plugin Developer's Guide

Steve Sylvester
Intel Corporation
steve.s.sylvester@intel.com

January 11, 2018

Abstract

Global Extensible Open Power Manager (GEOPM) is an extensible power management framework targeting high performance computing. The library can be extended to support new control algorithms and new hardware power management features. The GEOPM package provides built-in features ranging from static management of power policy for each individual compute node, to dynamic coordination of power policy and performance across all of the compute nodes hosting one MPI job on a portion of a distributed computing system. The dynamic coordination is implemented as a hierarchical control system for scalable communication and decentralized control. This document concentrates on the development of custom plugins for new control algorithms as well as new hardware platforms or new power management features.

1 Implementing new control algorithms

GEOPM supports two levels of control algorithms: a leaf control algorithm that manages power within a single node, and a tree control algorithm that manages power between nodes. Internally, these are known as tree deciders and leaf deciders.

2 Leaf Deciders

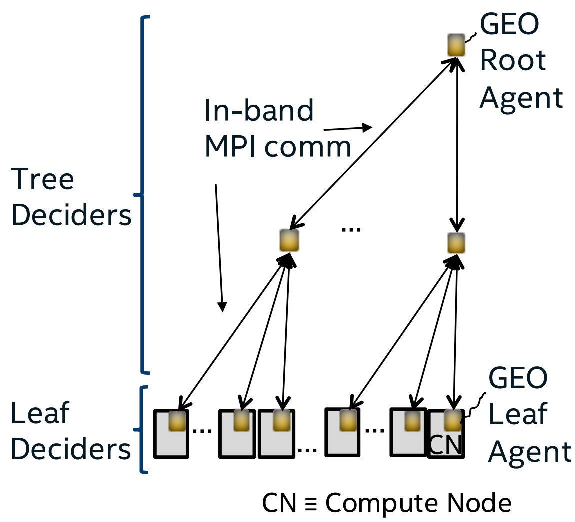

Figure 1: Leaf decider data collection

As seen in Figure 1, event counters and energy meters are collected from the platform while application data such as runtime and progress are collected from the individual application ranks on the node. This data is aggregated for each application region, where a region is a subset of the application runtime that has uniform computation characteristics. These characteristics include how compute-bound, memory-bound, or network-bound the region is. The leaf decider makes power decisions on a region by region basis. The leaf decider then uses this data to set new power policies for the domains of control on the particular hardware platform. For example, if the method of control is power capping then the domain of control would be the finest granularity hardware component with RAPL power controls. For current Xeon platforms based on Haswell, that would be at the socket level. If the method of control is frequency, then on current Xeon platforms based on Haswell, that would be at the CPU level. This data is abstracted away from the decider. It merely has per domain control statistics that it can use to set a power policy for the same set of domains. The current set of signals available to the leaf decider are captured in the `geopm_telemetry_type_e` enumeration:

```
enum geopm_telemetry_type_e {
    GEOPM_TELEMETRY_TYPE_PKG_ENERGY,
    GEOPM_TELEMETRY_TYPE_DRAM_ENERGY,
    GEOPM_TELEMETRY_TYPE_FREQUENCY,
    GEOPM_TELEMETRY_TYPE_INST_RETIRED,
    GEOPM_TELEMETRY_TYPE_CLK_UNHALTED_CORE,
    GEOPM_TELEMETRY_TYPE_CLK_UNHALTED_REF,
    GEOPM_TELEMETRY_TYPE_READ_BANDWIDTH,
```

```
    GEOPM_TELEMETRY_TYPE_PROGRESS,
    GEOPM_TELEMETRY_TYPE_RUNTIME,
    GEOPM_NUM_TELEMETRY_TYPE // Signal counter, must be last
};
```

3 Tree Deciders

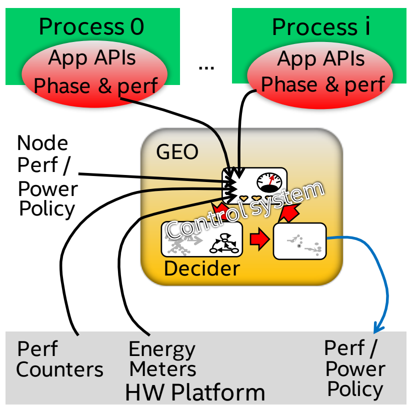

Figure 2: Data collection and power budget flow at the tree level

At the tree level, as can be seen in Figure 2, the domain of control is always the number of nodes which are direct children of the node in question. Data at the leaf level is aggregated and sent up the tree to the parent. That parent and its siblings have tree deciders, which use the aggregated data to create and send power budgets back down to the leaf nodes. The parent and its siblings also aggregate their data and send it up to the next level of the tree where the same thing happens. This continues all the way to the root of the tree. At any point in the tree the power budget is the aggregated budget for all leaf nodes in the nodes subtree. The current set of signals available to the leaf decider are captured in the `geopm_sample_type_e` enumeration:

```
enum geopm_sample_type_e {
    GEOPM_SAMPLE_TYPE_RUNTIME,
    GEOPM_SAMPLE_TYPE_ENERGY,
    GEOPM_SAMPLE_TYPE_FREQUENCY_NUMER,
    GEOPM_SAMPLE_TYPE_FREQUENCY_DENOM,
    GEOPM_NUM_SAMPLE_TYPE // Sample counter, must be last
};
```

4 Plugin Registration

Every plugin must export a C function that allows the plugin to register itself with the GEOPM runtime.

```
int geopm_plugin_register(int plugin_type, struct geopm_factory_c *factory,
                          void *dl_ptr)
{
    int err = 0;
    try {
        if (plugin_type == GEOPM_PLUGIN_TYPE_DECIDER) {
            geopm::Decider *decider = new geopm::CustomDecider;
            geopm_factory_register(factory, decider, dl_ptr);
        }
    }
    catch(...) {
         err = geopm::exception_handler(std::current_exception());
    }
    return err;
}
```

The function checks if it is the requested type, and if so, it creates an instance of the plugin class and registers it with the passed-in factory. In this example the plugin class is called `CustomDecider`.

5 Extending geopm::Decider base class

The `geopm::Decider` class has a number of pure virtual methods that must be overridden by the derived plugin class.

```
virtual Decider *clone() const = 0;
virtual const std::string& name(void) const = 0;
virtual bool decider_supported(const std::string &descripton) = 0;
virtual bool update_policy(Region &curr_region, Policy &curr_policy) = 0;
```

The `clone` method simply returns a copy of an instance of the class. This is for the internal use of the GEOPM runtime.

```
Decider *CustomDecider::clone(void) const
{
    return (Decider*)(new CustomDecider(*this));
}
```

A string description should be created for each plugin. The string should be unique among all decider plugins. If not unique then the first instance will be chosen at runtime. Some examples are:

```
const std::string m_name("powerbalancing");
```

or

```
const std::string m_name("powergoverning");
```

The name function simply returns the string description of the custom decider.

```
const std::string& CustomDecider::name(void) const
{
    return m_name;
}
```

The `decider_supported` method is the mechanism used by the runtime to choose a specific decider. It is simply a check against the string description of the plugin.

```
bool CustomDecider::decider_supported(const std::string &description)
{
    return (description == m_name);
}
```

The `update_policy` method is where the custom control algorithm is implemented. It takes in a `Region` object which holds the telemetry data for the current region as well as a `Policy` object which holds the current policy for all regions at this level of the tree. The `Region` object can be queried for telemetry for each domain of control through the following methods:

```
double signal(int domain_idx, int signal_type);
int num_sample(int domain_idx, int signal_type) const;
double mean(int domain_idx, int signal_type) const;
double median(int domain_idx, int signal_type) const;
double std_deviation(int domain_idx, int signal_type) const;
double min(int domain_idx, int signal_type) const;
double max(int domain_idx, int signal_type) const;
double derivative(int domain_idx, int signal_type) const;
```

All of these APIs take in a domain index, where the total number of domains can be queried from the `Policy` object. The second parameter is the signal type on which to operate. At the leaf they are `geopm_telemetry_type_e` enums and for the tree they are `geopm_sample_type_e enums`. The signal function returns the last recorded value of the requested signal type. The `num_sample` returns the number of samples used to calculate the values returned by the rest of the functions. The rest of the functions return the statistical value described by the function name over some number of samples. As a leaf-level example, to get the derivative of package power over the first domain of control:

```
bool CustomDecider::update_policy(Region &curr_region, Policy &curr_policy)
{
    int first_domain_index = 0;
    double pkg_power = curr_region.derivative(first_domain_index,
                                              GEOPM_TELEMETRY_TYPE_PKG_ENERGY);
. . .
```

At the tree level, to get the maximum runtime for the first domain of control:

```
bool CustomDecider::update_policy(Region &curr_region, Policy &curr_policy)
{
    int first_domain_index = 0;
    double max_runtime = curr_region.max(first_domain_index,
                                         GEOPM_SAMPLE_TYPE_RUNTIME);
. . .
```

After the statistics are used to calculate new power budgets or frequencies for each domain of control, the new budgets are applied to the policy object using either of the APIs:

```
void update(uint64_t region_id, int domain_idx, double target);
void update(uint64_t region_id, const std::vector<double> &target);
```

The Decider's `update_policy` function should return true if the policy was modified for any domain of control and false if no modifications were made. This signals the GEOPM runtime to apply the new budgets. When the algorithm reaches a state when it has figured out the best policy for the current region for the current power budget, the decider must mark the region as converged. This lets the GEOPM runtime know that it may send aggregated sample data up the tree to the parent, who then may send an updated power budget back down. This is done using the `Policy` object API:

```
void is_converged(uint64_t region_id, bool converged_state);
```

There is one more API that is not pure virtual but can be overridden by the plugin:

```
virtual bool update_policy(const struct geopm_policy_message_s &policy_msg,
                           Policy &curr_policy);
```

It is similar to the pure virtual `update_policy`, except that it takes in a `geopm_policy_message_s` structure instead of a Region object. The function gets called by the GEOPM runtime when it receives a new power policy message from its parent. At the root of the tree the parent would be a power-aware scheduler/resource manager. The default implementation simply splits the new budget evenly among all the domains of control, however different deciders could implement different methods of splitting the budget. One example is to split the new budget by the same ratios the current budget was split among the domains of control. Just like the previous `update_policy` function, it is expected that the function returns true if a policy is modified for any domain of control and false if no policies are modified. The budget in the `geopm_policy_message_s` structure is the total budget for all domains of control. It should be saved off as state for use in the pure virtual version of `update_policy`.

6 Implementing new platform plugins

The GEOPM package can be extended to run on new hardware platforms or support new power management features by creating `Platform` and `PlatformImp` plugins. A `PlatformImp` plugin implements the fine details of a specific hardware platform. In the case of an Intel platform, it encodes things such as the offsets of certain model-specific registers, as well as the topology and capabilities of that platform. A `Platform` plugin is an abstraction of the `PlatformImp` interface and is focused on methods of control that may be supported by many different `PlatformImp` objects. As an example, GEOPM has a built-in `RAPLPlatform` object which abstracts the power limiting functionality of Running Average Power Limiting (RAPL) as well as performance counter access across many different hardware platforms. The `PlatformImp` objects built into GEOPM include `IVTPlatformImp`, which supports Sandy Bridge- and Ivy Town-based Xeon systems, and a `HSXPlatformImp`, which supports Haswell-based Xeon systems. From the other perspective a `FreqPlatform` plugin could be created that modifies frequencies instead of RAPL power. It could use the same `PlatformImp` objects as the `RAPLPlatform` class without modifying them.

7 Plugin Registration

The plugin registration for a `Platform` or `PlatformImp` is similar to the `Decider` version. In fact, you can implement several plugins within the same file and have a single registration function:

```
int geopm_plugin_register(int plugin_type, struct geopm_factory_c *factory,
                          void *dl_ptr)
{
    int err = 0;
    Decider *decider = NULL;
    Platform *platform = NULL;
    PlatformImp *platform_imp = NULL;
    try {
        switch (plugin_type) {
            case GEOPM_PLUGIN_TYPE_DECIDER:
                decider = new CustomDecider;
                geopm_factory_register(factory, decider, dl_ptr);
                break;
            case GEOPM_PLUGIN_TYPE_PLATFORM:
                platform = new CustomPlatform;
                geopm_factory_register(factory, platform, dl_ptr);
                break;
            case GEOPM_PLUGIN_TYPE_PLATFORM_IMP:
                platform_imp = new CustomPlatformImp;
                geopm_factory_register(factory, platform_imp, dl_ptr);
                break;
        }
    }
    catch(...) {
        err = exception_handler(std::current_exception());
    }
    return err;
}
```

8 Extending geopm::Platform base class

To extend the `geopm::Platform` class, you must override the following pure virtual functions:

```
virtual  bool model_supported(int platform_id, const std::string &description)
const = 0;
virtual  size_t capacity(void) = 0;
virtual  void sample(std::vector<struct geopm_msr_message_s> &msr_values) = 0;
virtual  void enforce_policy(uint64_t region_id, Policy &policy) const = 0;
```

The `model_supported` function is the mechanism used by the runtime to choose a specific platform. A hardware identifier as well as a description string are passed in by the runtime. The plugin uses these as well as any other available information in order to decide if it sufficiently supports the requested functionality as well as the underlying hardware. The GEOPM runtime uses the cpuid instruction to get the hardware platform ID and sends it in as the first parameter. The passed-in description string comes from the global power policy. For example, the built-in `RAPLPlatform` class is implemented as follows:

```
const int M_SNB_ID = 0x62D; // SandyBridge E
const int M_IVT_ID = 0x63E; // IvyTown E
const int M_HSX_ID = 0x63F; // Haswell E
const std::string m_description("rapl");
bool RAPLPlatform::model_supported(int platform_id,
                                   const std::string
                                   &description) const
{
    return ((platform_id == M_IVT_ID ||
             platform_id == M_SNB_ID ||
             platform_id == M_HSX_ID) &&
            description == m_description);
}
```

The `capacity` method returns the number of signals that will be returned when the sample method is called. This lets the caller pre-size the vector that will hold the data. For the `RAPLPlatform` it returns 3 signals per socket and 4 signals per CPU and is implemented as follows:

```
size_t RAPLPlatform::capacity(void)
{
    return m_imp->num_domain(m_imp->power_control_domain()) *
        (m_imp->num_energy_signal() + m_imp->num_counter_signal());
}
```

The `m_imp` is a pointer to a `PlatformImp` instance which abstracts the specifics of the underlying hardware. The `sample` method takes in a pre-sized vector and fills it with sampled data. It gets the samples from the `PlatformImp` object that knows how to query the hardware. It uses the following API from the PlatformImp object:

```
double read_signal(int device_type, int device_index, int signal_type);
```

Filling in values for package, dram, and power plane 0 energy on all sockets:

```
void RAPLPlatform::sample(std::vector<struct geopm_msr_message_s> &msr_values)
{
```

```
    int count = 0;
    struct geopm_time_s time;
    geopm_time(&time);
    //record per package energy readings
    for (int i = 0; i < m_num_package; i++) {
        msr_values[count].domain_type = GEOPM_DOMAIN_PACKAGE;
        msr_values[count].domain_index = i;
        msr_values[count].timestamp = time;
        msr_values[count].signal_type = GEOPM_TELEMETRY_TYPE_PKG_ENERGY;
        msr_values[count].signal =
            m_imp->read_signal(GEOPM_DOMAIN_PACKAGE, i,
                               GEOPM_TELEMETRY_TYPE_PKG_ENERGY);
        count++;
        msr_values[count].domain_type = GEOPM_DOMAIN_PACKAGE;
        msr_values[count].domain_index = i;
        msr_values[count].timestamp = time;
        msr_values[count].signal_type = GEOPM_TELEMETRY_TYPE_DRAM_ENERGY;
        msr_values[count].signal =
            m_imp->read_signal(GEOPM_DOMAIN_PACKAGE, i,
                               GEOPM_TELEMETRY_TYPE_DRAM_ENERGY);
        count++;
    }
. . .
```

The `enforce_policy` function takes in a region ID and `Policy` object. It must then enforce the policy for the specified region id encoded in the `Policy` object. It uses the `PlatformImp` method:

```
void write_control(int device_type,
                   int device_index,
                   int signal_type,
                   double value);
```

For the RAPLPlatform to set new power budgets for package level power:

```
void RAPLPlatform::enforce_policy(uint64_t region_id, Policy &policy) const
{
    int control_type = GEOPM_TELEMETRY_TYPE_PKG_ENERGY;
    std::vector<double> target(m_imp->power_control_domain());
    policy.target(region_id, target);
    for (int i = 0; i < m_num_package; ++i) {
        m_imp->write_control(m_imp->power_control_domain(), i,
                             control_type, target[i]);
    }
}
```

9 Extending geopm::PlatformImp base class

To extend the `geopm::PlatformImp class`, you must override the following pure virtual functions:

```
virtual std::string platform_name(void) = 0;
virtual bool model_supported(int platform_id) = 0;
virtual void msr_reset(void) = 0;
virtual void msr_initialize() = 0;
virtual int power_control_domain(void) const = 0;
virtual int frequency_control_domain(void) const = 0;
virtual double read_signal(int device_type,
                           int device_index,
                           int signal_type) = 0;
virtual void write_control(int device_type,
                           int device_index,
                           int signal_type,
                           double value) = 0;
```

A string description should be created for each plugin. The string should be unique among all `PlatformImp` plugins. If not unique then the first instance will be chosen at runtime. For example:

```
const std::string m_name("Haswell E");
```

The `platform_name` method simply returns the string name of the platform.

```
std::string HSXPlatformImp::platform_name()
{
    return m_name;
}
```

The `model_supported` method is the mechanism used by the runtime to choose a specific platform implementation. A hardware identifier is passed in by the runtime. The plugin uses this as well as any other available information in order to decide if it sufficiently supports the requested functionality as well as the underlying hardware. The GEOPM runtime uses the cpuid instruction to get the hardware platform id and sends it in as the first parameter. For example, the built-in `HSXPlatform` class is implemented as follows:

```
const in m_platform_id = 0x63F; // Haswell E
bool HSXPlatformImp::model_supported(int platform_id)
{
    return (platform_id == m_platform_id);
}
```

The `msr_reset` function should clear any programable counters or RAPL MSRs that it may have been modified during runtime. An example of this would be clearing out any RAPL power limits. The `msr_initialize` method should do any initialization of performance counters or RAPL MSRs that will be needed during runtime. The `power_control_domain` method should return a `geopm_domain_type_e` enumeration of the finest granularity hardware that supports RAPL power limiting; for Haswell, this is at the package level.

```
int HSXPlatformImp::power_control_domain(void) const
{
    return GEOPM_DOMAIN_PACKAGE;
}
```

The `frequency_control_domain` method should return a `geopm_domain_type_e` enumeration of the finest granularity hardware that supports independent frequency scaling; for Haswell, this is at the CPU level.

```
int HSXPlatformImp::power_control_domain(void) const
{
    return GEOPM_DOMAIN_CPU;
}
```

The `read_signal` method is tasked with returning a single sampled value from the underlying hardware. A `geopm_domain_type_e` enum is passed in identifying the device type to query as well as an index of which device of that type to query. The last parameter is the signal type being requested. In general there should be a case handler for each `geopm_telemetry_type_e` enumerated type.

```
double HSXPlatformImp::read_signal(int device_type, int device_index,
                                   int signal_type)
{
    double value = 0.0;
    int offset_idx = 0;
    switch (signal_type) {
        case GEOPM_TELEMETRY_TYPE_PKG_ENERGY:
            offset_idx = device_index * m_num_package_signal;
            value = msr_overflow(offset_idx, 32,
                                 (double)msr_read(device_type,
                                                  device_index,
                                                  m_signal_msr_offset[0]));
            value *= m_energy_units;
            break;
        case GEOPM_TELEMETRY_TYPE_DRAM_ENERGY:
            offset_idx = device_index * m_num_package_signal + 2;
            value = msr_overflow(offset_idx, 32,
                                 (double)msr_read(device_type,
                                                  device_index,
                                                  m_signal_msr_offset[2]));
            value *= m_energy_units;
            break;
        case GEOPM_TELEMETRY_TYPE_FREQUENCY:
            offset_idx = m_num_package * m_num_package_signal +
                         device_index * m_num_cpu_signal;
            value = (double)(msr_read(device_type,
                                      device_index / m_num_cpu_per_core,
                                      m_signal_msr_offset[3]) >> 8);
            //convert to MHZ
            value *= 0.1;
            break;
. . .
```

The `write_control` method is tasked with writing a single value from the underlying hardware control. A `geopm_domain_type_e` enum is passed in identifying the device type to query as well as an index of which device of that type to query. The second to last parameter is the signal type being requested. The last parameter is the value to set. In general, there should be a case handler for each `geopm_telemetry_type_e` enumerated type that is controllable.

```
void HSXPlatformImp::write_control(int device_type, int device_index,
                                   int signal_type, double value)
{
    uint64_t msr_val = 0;
    switch (signal_type) {
        case GEOPM_TELEMETRY_TYPE_PKG_ENERGY:
            if (value < m_min_pkg_watts) {
                value = m_min_pkg_watts;
            }
            if (value > m_max_pkg_watts) {
                value = m_max_pkg_watts;
            }
            msr_val = (uint64_t)(value * m_power_units);
            msr_val = msr_val | (msr_val<<32) | M_PKG_POWER_LIMIT_MASK_MAGIC;
            msr_write(device_type, device_index,
                      m_control_msr_offset[0], msr_val);
            break;
        case GEOPM_TELEMETRY_TYPE_DRAM_ENERGY:
            if (value < m_min_dram_watts) {
                value = m_min_dram_watts;
            }
            if (value > m_max_dram_watts) {
                value = m_max_dram_watts;
            }
            msr_val = (uint64_t)(value * m_power_units);
            msr_val = msr_val | (msr_val<<32) | M_DRAM_POWER_LIMIT_MASK_MAGIC;
            msr_write(device_type, device_index,
                      m_control_msr_offset[2], msr_val);
            break;
        case GEOPM_TELEMETRY_TYPE_FREQUENCY:
            msr_val = (uint64_t)(value * 10);
            msr_val = msr_val << 8;
            msr_write(device_type, device_index / m_num_cpu_per_core,
                      m_control_msr_offset[3], msr_val);
            break;
. . .
```

10 Plugin installation

Plugins need only to be put into the geopm plugin directory. This is typically in your systems library directory inside a geopm directory. For instance, on Redhat and CentOS plugins are located within `/usr/lib64/geopm/`. If you do not have permissions to write to that directory, you may place them in another location pointed to by the `GEOPM_PLUGIN_PATH` environment variable. Note that plugins are processed before built-in classes making it easy to override defaults with a custom plugin.

11 Copyright and disclaimer