White Paper on PowerStack

Andrea Bartolini, Stephanie Brink, Daniele Cesarini, Dan Ellsworth, Ryan Grant, Siddhartha Jana, Masaaki Kondo, Eun Kyung Lee, Matthias Maiterth, Aniruddha Marathe, Tapasya Patki, Swann Perarnau, Valentin Reis, Martin Schulz, Ondrej Vysocky, Torsten Wilde, Xingfu Wu

June 12, 2020

List of Contributors (in alphabetical order)

- Andrea Bartolini (University of Bologna, Italy)
- Stephanie Brink (Lawrence Livermore National Laboratory, USA)
- Daniele Cesarini (CINECA, Italy)
- Dan Ellsworth (Colorado College, USA)
- Ryan Grant (Sandia National Laboratories, USA)
- Siddhartha Jana (Energy Efficient HPC Working Group)
- Masaaki Kondo (The University of Tokyo & RIKEN, Japan)
- Eun Kyung Lee (IBM, USA)
- Matthias Maiterth (Ludwig-Maximilians University, Germany)
- Aniruddha Marathe (Lawrence Livermore National Laboratory, USA)
- Tapasya Patki (Lawrence Livermore National Laboratory, USA)
- Swann Perarnau (Argonne National Laboratory, USA)
- Valentin Reis (Argonne National Laboratory, USA)
- Martin Schulz (Technical University Munich, Germany)
- Ondrej Vysocky (IT4Innovations National Supercomputing Center, Czech Republic)
- Torsten Wilde (HPE, USA)
- Xingfu Wu (Argonne National Laboratory, USA)

1. Introduction

1.1 Background and Motivation of Designing an HPC PowerStack

(Contributor: Sid, Martin, Masaaki / Target #pages = more than 3pages)

The landscape of High-Performance Computing (HPC) is changing as we enter the exascale era and power and energy management are key design points for any next generation of supercomputers. Efficiently utilizing procured power and optimizing performance of scientific applications under power and energy constraints is challenging due to several reasons including dynamic phase behavior, processor manufacturing variability and increasing heterogeneity of node-level components. Extending the scope from the node- and application-level up to the system-level introduces further challenges on top of power-unaware job scheduling, which is known to be NP-hard on its own [4].

While there exists several individual efforts across the community to research automatic techniques for managing power and energy better, the majority of these techniques have been specialized to meet the needs of a specific HPC center or specific optimization goals and provide little support to connect them to each other. Specifications such as PowerAPI [5,6] and Redfish [8] provide high-level power management interfaces for accessing power knobs. However, these stop short of defining which software components should actually be involved, and how they interoperate with each other in a cohesive and coordinated stack. We believe coordination is critical for avoiding underutilization of system Watts and FLOPS. These efforts form a good starting point for full stack integration, but these interfaces still need to be hooked up to the wide range of software components offered by academic partners, developers or vendors. Furthermore, a recent survey [3] conducted by the EEHPC WG [7] concluded that the majority of such techniques have lacked the application-awareness required to achieve the best system performance and throughput. Other observations were that each technique tended to target management of power and energy for a different subset of the site or system hardware and that the different techniques tended to perform management at different (and often conflicting) granularities. Unfortunately, the existing techniques have not been designed to coexist simultaneously on one site and cooperate on management in an integrated manner. The lack of application-awareness, lack of coordinated management across different granularities, and the lack of widely accepted interfaces and consequent limited connectivity between modules, can result in substantially underutilized Watts and FLOPS.

To address these gaps, the HPC community needs a holistic stack for power and energy management and none currently exists. In our view, a holistic stack is one that is extensible enough to support the present and anticipated future needs of various different HPC centers, one that achieves best system performance and throughput through application-awareness, one that is designed to coordinate management at different granularities, and one that enables the seamless integration of software components from different developers and vendors.

In this white paper, we provide a high-level overview of a software stack which manages power and energy of HPC systems and standardized interfaces between different levels of software components in the stack.

1.2 Target and Scope of the HPC PowerStack

(Contributor: Andrea, Tapasya)

The PowerStack initiative started in May 2018 as a workgroup to gather the experience of active actors in industry, computing centres and academia in building software interfaces and solutions for handling and optimizing the power and energy consumption in HPC systems in production. The target of the PowerStack initiative is to collect and define: (i) the common technologies needed for controlling effectively the power and energy consumption at scale in an HPC environment; (ii) the software abstractions for connecting these common technologies with run-times, applications, system management tools; (iii) the use-cases and scenarios requirements supported by the PowerStack.

The PowerStack does not aim to impose a given technology but rather standardize the best-practices and HW/SW interfaces already made available or planned by vendors, funded initiatives and researchers in a common direction toward power & energy self-aware and sustainable HPC data centres.

We expect the following target audiences of this white paper.

- Administrator of HPC centers responsible of procuring HPC systems
- HPC vendors who implement HPC systems
- Researchers developing power management tools and algorithms
- Hardware and software architects of HPC systems

1.3 Relationship with Other Community Wide Efforts

(Contributor: Ryan, Sid)

The HPC PowerStack effort has deep connections to other efforts in the power/energy management and control.

One such synergistic effort is the PowerAPI specification community [5,6] that defines high-level interfaces between different levels of the PowerStack as well as low level power/energy device interface abstraction using a put/get model. Informational metadata about measurement points and key data relating to the methods of collection, accuracy and underlying sampling rates can also be obtained through PowerAPI interfaces at any level, allowing for portable tools and management software to be written as well as convenient interfaces for portability for job managers and other optimizing runtimes. The communities of the PowerStack and PowerAPI overlap significantly as the goals of each community are complementary.

Low-level vendor specifications like IPMI [9] and Redfish [8] enable remote control of computer systems via out-of-band and/or side-band communication channels. The power management specific controls enabled by these specifications constitute a very small subset of a larger group of remote control capabilities. However, by design, these interfaces operate through the BMC and a LAN connection via a NIC, and are therefore limited by bandwidth capabilities. Moreover, these specifications cater to a very limited set of power-management capabilities and leveraged by system-admins for privileged access to the systems. This limits opportunities for leveraging user-level application-awareness within the HPC software stack.

2. PowerStack Specification and Features

(Target #pages = unlimited)

The PowerStack operates on and across the logical and physical planes of the system. The logical plane including cluster, job, node process, and thread, organizes the computing elements of the HPC system by the job assigned to the computing elements. Considering an HPC system from the perspective of the logical plane provides an application centered view of system operation that abstracts away much of the physical details of the system. The physical plane including data center, computer room, row, rack, node, socket, and cores, organizes the computing element of the HPC system by their location within the machine room and connected physical infrastructure and abstracts away much of the details of the computational work being done. Considering an HPC system from the perspective of the physical plane provides an infrastructure centered view of system operation. PowerStack enables measurement, control, and correlation across physical and logical contexts by bridging the logical and physical planes of the system.

(ToDo: abstract picture)

Current PowerStack conceptualizes the problem in general as a hierarchical system in logical and physical planes to optimize power, energy and performance of the system. For example, we may define a scope of the stack working at the node level, job level, and system level. In this section, we will specially focus on these levels and show the goal driven interaction between software agents at these levels as the basis of PowerStack.

[Revised based on the comment, see above] Current PowerStacks conceptualize the problem as a three tiered system working at the node level, job level and system level, optimizing for power, energy and performance of the system. The goal driven interaction between software agents at these levels are the basis for PowerStack.

TODO: Sid - add the PS block diagram from June’19 seminar

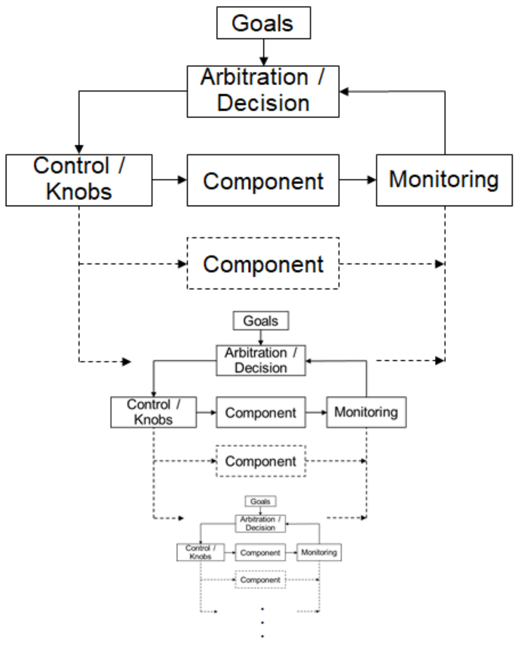

Figure 2.1 Hierarchical nature of PowerStack

2.1 Definition of the HPC PowerStack

(Contributor: Matthias, Dann, Sid)

Specification of functionality of a hardware and software stack to manage power and energy in an HPC environment using schedulers, runtime systems and node management capabilities.

A PowerStack provides well defined capabilities for controlling and monitoring power, energy, and thermal concerns within a HPC system at the granularity required by the HPC center. Due to the diverse needs and objectives of different HPC centers, a single one size fits all PowerStack implementation is not practical and likely not desirable. Using a broad definition admits a number of solutions that can be tailored and potentially mixed for specific cost and deployment contexts.

In creating the PowerStack specification, four primary system goals an operator might try to control for should be considered. These goals are interrelated and form a complex multiobjective optimization space that a HPC system exists within. Changes in the system configuration to meet one objective will likely impact the other three goals.

- **Power** – rate of incoming electrical energy to a system on short time scales. Short refers to time scales that would not cause a breaker to trip or beyond what the internal hardware can smooth.
- **Energy** – aggregate power consumed over an interval. The interval could be a billing interval or energy to completion type metric.
- **Thermal** – rate of outgoing thermal energy from a system on short time scales.
- **Time to Solution** – the time taken to complete a given computation on a given input size. For example, a weather prediction computation that does not complete before the weather actually occurs has limited value.

The PowerStack specification spends the least time on the Time to Solution metric. This is the classically dominant HPC metric and existing tools and techniques are very effective in this space. The newer and less well supported operational goals are meeting power targets, energy targets, and thermal targets. Achieving control and monitoring over these three targets, without losing sight of time to solution, is anticipated to be the core value add of adopting PowerStack within an HPC system.

A PowerStack that provides well defined control and monitoring capabilities is considered **compliant**. Compliance provides an assurance that components are able to provide verifiable capabilities related to control and monitoring. A PowerStack is **conformant** when the control and monitoring capabilities operate within the latency, component scope, and granularity required for operation of the system. Determining conformance requires selection of a constellation of PowerStack compliant components and defining the environment, if the selected components are able to monitor and control with satisfactory latency and granularity the PowerStack is conformant.

Each PowerStack compliant component must specify the **latency**, **component scope**, and **granularity of monitoring and control** for each metric the PowerStack component is responsible for. Installations in which PowerStack components are stacked must guarantee that lower layers in the stack conform to the requirements of layers higher in the stack. For example, a system wide power controller should not claim a 100ms latency to setting system power if the node level controllers it relies on to execute that action have a 500ms latency.

- **Latency** is defined as the wall clock time between when a PowerStack component sends a request to another actuator component and when the actuator component receives the request or between when a PowerStack component receives a request for an action and when the actuation is complete in a measurable way, or sum of them including any delays (e.g. queuing delays). [Revised based on the comment]

(guid minimum latency, each layer should guarantee the minimum numbers
-> range of exact number will be given in the following sections)

- **Component scope** is defined as the physical (e.g. node) and logical (e.g. job) region of the system the component is responsible for and the metrics the component captures/controls.
- **Granularity of monitoring and control** is defined by the temporal resolution and fineness of setting/measurement of the metrics as well as the error.

Conflicting Goal Resolution

When defining the operating environment for conformance, three levels of constraint criticality should be considered in relation to each goal.

- **Hard constraint** – this severity level refers to constraints that would result in a physical loss/damage to the machine or catastrophic damage to the organization if exceeded. Hard constraints are non-negotiable limits and must be enforced by the system at all times. As an example, exceeding the power capacity of the power distribution infrastructure.
- **Soft constraints** – this severity level refers to constraints that are important goals for the system and might be costly to the organization if not met. As an example, slightly

exceeding the contracted energy with a power provider would not harm the system but might result in uncomfortable budgetary challenges.

- **Wishes** – this severity level refers to constraints that are useful for research or other exploration, for which missing the goal has negligible impact to the organization.

The same goal may simultaneously be set at multiple severity levels. Interactions between severities is hierarchical:

- A hard constraint always takes precedence over a soft constraint
- A soft constraint always takes precedence over a wish

When merging hard constraints, the most restrictive hard constraint must be obeyed. For example, to satisfy a hard thermal constraint the power may be lowered well below the hard power constraint. Optimization is an important, but secondary, concern for hard constraints due to the extreme cost of missing the goal.

Soft constraints allow more flexibility in considering goal prioritization as a multivariable optimization problem. Powerstack components implementing performing optimizations should provide guidance on how conflicting goals and situations in which not all objectives can be met are resolved.

Wishes, to an extent, can be considered informational to the system. So long as the wish does not conflict with the hard and soft constraints, the setting can be granted. If conflicts exist with hard or soft constraints, it is expected that the system will reject or only partially fulfil the wish so that the higher priority hard and soft constraints continue to be maintained. The system granting the wishes should provide guidance in how conflicts are resolved.

2.2 Necessary functionalities for PowerStack

PowerStack requires mandatory and optional functionalities for each tier (node-, job-, and system-level) controllers of the PowerStack. Since the specific features for those functionalities may vary depending on the hardware or/and software supports (and some are vendor specific), this section only highlights the high-level mandatory and optional functionalities of PowerStack controllers.

2.2.1 Mandatory Functionalities

(Contributor: Matthias, Sid, EK)

Table 2.1 summarizes mandatory functionalities of PowerStack for each tier.

Table 2.1: Mandatory functionalities of PowerStack.

PowerStack Tier	Monitoring	Control/Actuation
Node Level	- Power and energy usage (Computing units, memory, IO and storage, and node)	- Different Power State - Power capping (computing units, memory)

System Level	- Power and energy usage (Distributed Jobs) - System level software to monitor current job behavior (e.g. signals and telemetry describing application and platform characteristics)	- Dynamic job allocation (scheduler) aiming energy-aware scheduling - System-level software that can interact with job-level actuator
Job Level	- Power and energy usage	- Power capping/budgeting per job - Mechanisms to leverage monitored information to drive changes in application and node performance

Note that providing information about temporal/spatial granularity of the measurement is also a mandatory requirement.

2.2.2 Optional Functionalities

In order to efficiently utilize available power and energy for computing in HPC systems, we can define several functionalities which are desirable to be implemented in the HPC PowerStack. The followings are examples of such *optional* functionalities.

- Temporal/spatial granularity of the measurement, however we recommend XX [Please help to define them]
- Controllability of temporal/spatial granularity of the measurement.
- Power/energy measurements that are not mentioned in the mandatory functionalities. (e.g., network devices, UPC, cooling)
- Optimizing resolution/accuracy of measurement and control
- Processor/hardware specific power management features (e.g., P, S, C-state, Turbo-mode, etc)
- Broader scope of power management (e.g., rack, zone- level power capping and monitoring)

(Contributor: Andrea, EK)

2.2.3 API Design

(Contributor: Ryan)

A key component in any power/energy management software stack is a portable design as differences between vendor implementations can be very large. Insulating users from changes in underlying power management hardware is important to the development of a rich ecosystem of tools, power aware scheduling and job management. The need for a portable API for probing device information, gathering samples and changing device settings exists throughout the PowerStack. As such a portability layer is key to the viability of a

PowerStack. However, there are considerations with respect to the underlying engineering aspects of a PowerStack. Therefore, high level portable interface design is important but should not be tied to the underlying technologies used to enact the interface's implementation. The PowerAPI [5,6] provides a portable interface and has been in development for several years, both as a specifications document as well as a reference implementation. The PowerAPI specifies interfaces for many levels of a data center/supercomputer spanning all of the levels of the PowerStack. As such it is a reasonable API to use in interactions between layers of the PowerStack that are not a single integrated product, for example a job scheduler interacting with a job optimizing runtime. PowerAPI can also be used within a given layer of the PowerStack if desired. For example, a job scheduler might use the PowerAPI to communicate a power cap for a job to a job management layer, but also use the PowerAPI to set the hard power cap for the nodes or cabinets that the job is executing on.

PowerAPI defines levels of the overall system described in Figure 2.2 detailing different potential users/actors for the portable interface [5,6]. Such actors may or may not be human users, several points throughout the system may be entirely automated.

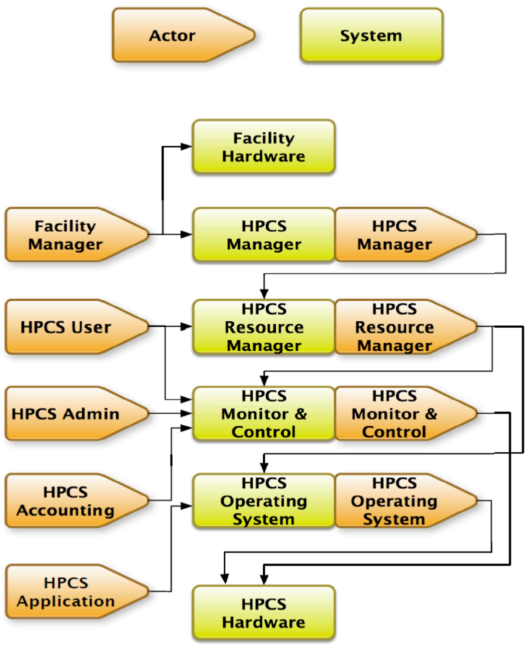

Figure 2.2: PowerAPI Actor Designations

In addition to the description of the actors that may interact with the PowerAPI, the portability of the description of the system is based on state of the art hardware description approaches that designate different levels of the system hardware in a hierarchical manner. Figure 2.3 shows an example of a hierarchical system description with a subset of available PowerAPI objects.

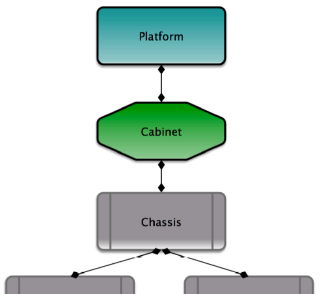

Figure 2.3 Hierarchical system objects description

2.3 A Strawman Design

(Contributor: Daniele, Masaaki)

One of the key aspects of Powerstack is to define an envision of a holistic power and energy management stack that could be extensible and application-aware to be capable to trade-off power, energy and time-to-solution to optimize the efficiency of an HPC application. The second aspect is to define a standard interface to interact with optimization SW and HW knobs across different vendor HPC systems.

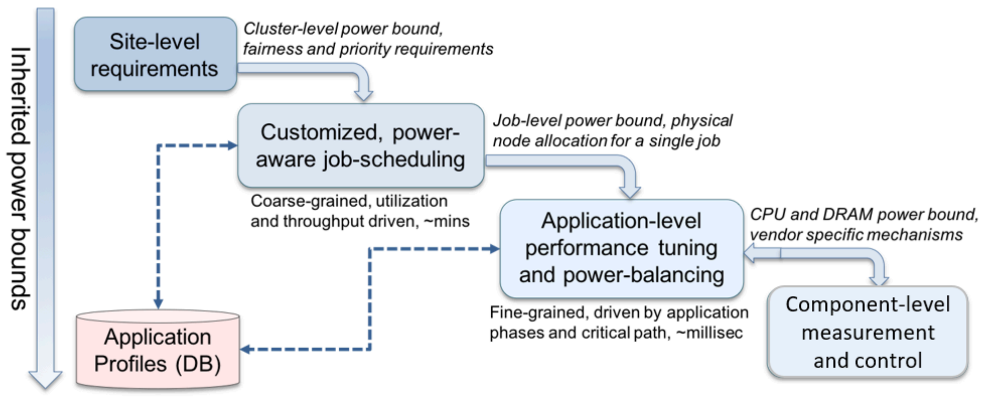
Inherited power bounds
Site-level requirements
Cluster-level power bound, fairness and priority requirements
Customized, power-aware job-scheduling
Job-level power bound, physical node allocation for a single job
Coarse-grained, utilization and throughput driven, ~mins
Application-level performance tuning and power-balancing
CPU and DRAM power bound, vendor specific mechanisms
Fine-grained, driven by application phases and critical path, ~millisec
Component-level measurement and control
Application Profiles (DB)

Figure 2.4 A strawman PowerStack design

In the state-of-the-art works available in the HPC power and energy management community [5,6,10], there are a multitude of systems that considers a wide range of interaction levels that can shape energy and power of a system. Based on the state-of-the-art works, we propose a strawman PowerStack design which implements a hierarchical HPC concept to manage power and energy based on three levels of granularity as shown in Figure 2.4.

- **System level**: At the system level, it is possible to define power/energy bounds, job priorities and fair policies that will be propagated in the job level. The system level is usually represented by the job scheduler of the system that can be extended through power-aware scheduling plugins to ensure the compliance of Powerstack. The main responsibility of the job scheduler is the allocation of the nodes and the power constraints across multiple jobs. Such constraints will serve as inputs to different jobs to tune the power control of the compute nodes.

- **Job level**: The job level is responsible to coordinate optimizations of compute node hardware control settings across the compute nodes in a job. The job level relies on node-level control mechanisms and telemetry. Continuous monitoring and analysis of application behavior in the compute node is required for decision-making.

- **Node level**: The node level describes the power knobs that each compute node must implement to fine-grain tune power, energy and performance of the application instances that are running on that node. Each compute node must be managed according to the job level, but the top-level constraints must be employed across the nodes of the system simultaneously.

Based on the high-level abstraction laid out above, in the following sections we will describe how Powerstack concrete implements those abstraction levels.

3. Use Cases and Research Efforts

(Target #pages = typically 1-3 pages and no more than 5 pages each)

3.1 Use Cases of the HPC PowerStack

(Contributor:Torsten)

Use-cases are the fundamental drivers for the PowerStack effort. They are unifying on a high-level but the exact control policy can be very site specific. For example, the requirement to keep a data center operating below a certain power bound could be imposed by budgetary constraints or alternatively by the limitations of the cooling infrastructure or power supply. These cases fall under the power capping use-case but the enforcement might have different tradeoffs and requirements.

Following definitions are used:

- System Manager: Vendor provided control software allowing the management of a system out-of-band (for example, HPE HPCM)
- Job scheduler: Responsible for allocation resources to jobs, batching jobs, and launching jobs
- Job management runtime: Responsible for keeping a job inside a defined power envelop
- Node: Assuming exclusive resource usage, the smallest integrated hardware resource available to a job to run on (usually consists of compute units (like CPUs or GPUs), memory, disk, power regulation components)

Some of the fundamental use-cases are covered in the following subsections.

3.1.1 Keep my system under a power cap

This use-case enables a site to set a system power cap. PowerStack will ensure that system power stays below the power cap.

For this use-case, a system power cap is defined and set by, for example, the site administrator (Figure 3.1 left). The power cap is provided to the *Job scheduler* which splits the system budget into individual job budgets. The *Job management runtime* enforces the job power budget at the job and consecutively on the node level. We can call this “In-band” enforcement.

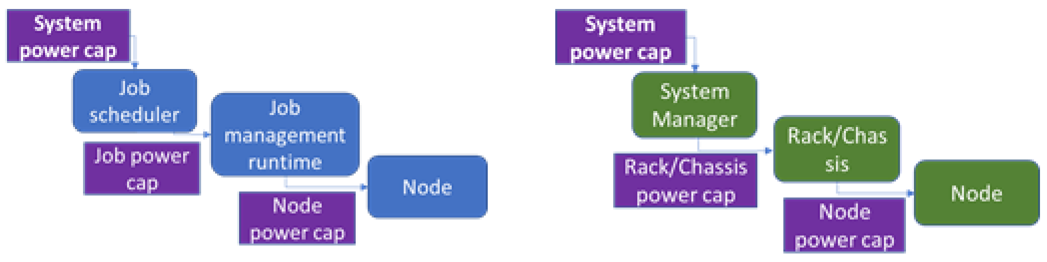

Figure 3.1 Use cases of Power cap enforcement

A variation of this use-case is the setting of static node power caps by a *System manager* (Figure 3.1 right). If the *System manager* is not job power aware, it will split the system power cap into even power caps for all the nodes (and/or potentially for hardware based system partitions such as *racks* or *chassis*). It then uses its management infrastructure to set *node* power caps. If those caps are enforced by the node hardware (for example, board management controller) we call this "out-of-band" enforcement.

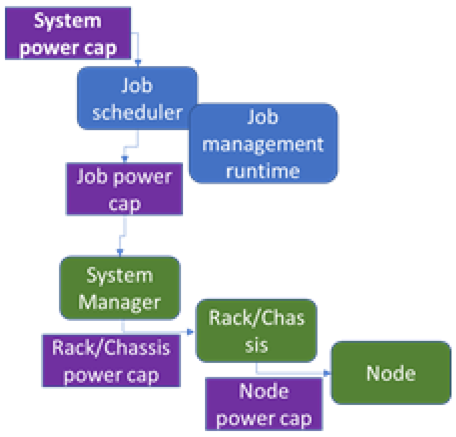

Figure 3.2 Combination of in-band and out-of-band power cap enforcement

Another variation is the combination of "in-band" and "out-of-band" enforcement. Here, the system power cap is, for example, provided to the *Job scheduler* which splits the system budget into individual job budgets. The job budgets (together with the job node list) is sent to the *System manager*. The *System manager* splits the job budget into even budgets for each *node* of the job. It then uses its management infrastructure to set *node* power caps which are enforced by the node hardware.

3.1.2 Optimize Energy-to-Solution (EtS)

This use-case looks into minimizing the cost of running an application in terms of energy consumption by optimizing the energy consumed by a particular job. This will typically be based on a trade-off between jop runtime and total energy consumption by tweaking certain performance parameters that influence the power consumption of a node (e.g., frequency/voltage of CPU cores, uncore, RAM, GPU, etc).

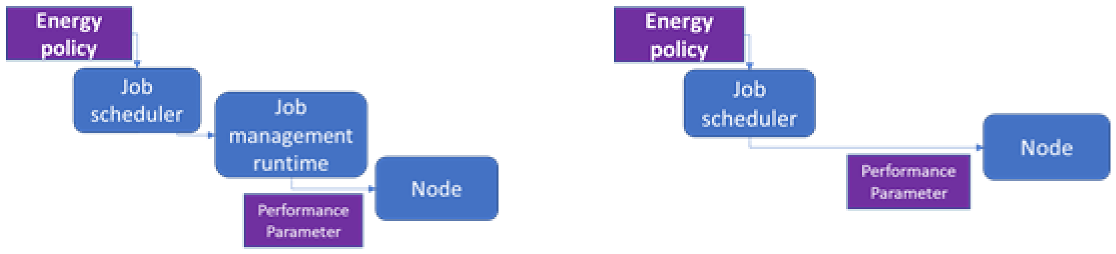

Figure 3.3 Dynamic (left) and static (right) use cases of minimizing energy-to-solution

This can either be implemented via a dynamic or a static algorithm. In the dynamic use-case the *Job scheduler* gets the policy definition and passes it on to the *Job management runtime* which then varies the *node* performance parameters at runtime in order to fulfull the objectives of the policy.

For the static control, the energy optimization policy is also passed to the *Job scheduler* but now the *Job scheduler* sets the *node* performance parameters only once at the beginning of the job in order to fulfill the objectives of the policy.

3.1.3 Optimize system-level job throughput under a power budget

This use-case focuses on whole system job throughput under a power cap rather than optimizing individual job performance.

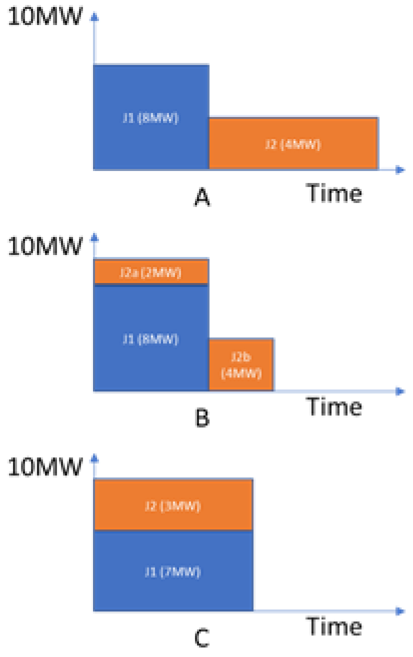

Figure 3.4 Use cases of optimizing system-level job throughput under a power budget

The assumption is that there is a power cap in effect. Without further action, jobs that might have run in parallel on the unconstrained system are now spread out temporally since their combined unmanaged power consumption would be higher than the imposed power limit (Figure 3.4-A).

However, by knowing the runtime and power trade-off for each application (through offline benchmarking and power profiling, or similar), a new set of more optimal, intra-application runtime states can be found. These states allow for system-level throughput optimization by either: (1) reducing the power of application J2 for the duration of application J1 (shown as J2a) providing a marked throughput improvement (Figure 3.4-B), or (2) by reducing the power of both applications J1 and J2 and thereby finishing even earlier than in the first case (Figure 3.4-C).

For this use-case, detailed application characteristics (such as power, performance, runtime, node count etc.) need to be collected and analyzed. Job runtimes could potentially communicate with each other to trade power budgets for specific time intervals. This use-case combines the need for a system-level power cap with potentially dynamic job power budgets.

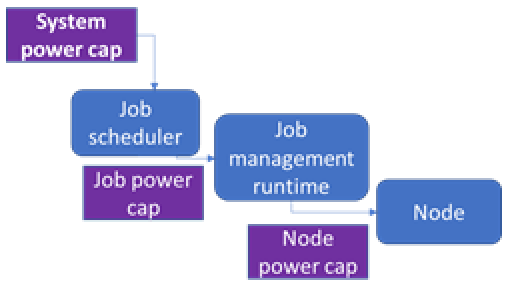

Figure 3.5 Flow of power cap enforcement in optimizing job throughput

A system power cap is transmitted to the *Job scheduler*. The *Job scheduler* uses application characteristics to define the optimal job mix under the power cap. It then provides the job power cap to the *Job management runtime*. The *Job management runtime* sets specific *node* power caps that are enforced by each *node*.

3.2 Research Efforts for PowerStack Interoperability

3.2.1 GEOPM

(Contributor: Sid)

GEOPM (Global Extensible Open Power Manager) is an active, community-driven, cross-platform, open source (BSD 3-clause), job-level power-management framework that is capable of providing feedback to job schedulers and resource managers in order to drive system power efficiency improvements (Figure 3.6). This framework is part of OpenHPC and is a key pillar of the software stack in the upcoming first exascale system at Argonne National Laboratory in the US. It optimizes for time-to-solution by leveraging techniques from learning and control systems. GEOPM leverages hardware monitoring and control knobs provided by Intel's hardware power limiting capability called RAPL. Additionally, other processor architectures, such as IBM POWER and ARM have recognized the value of GEOPM and are currently working to add their specific processor support to the GEOPM stack.

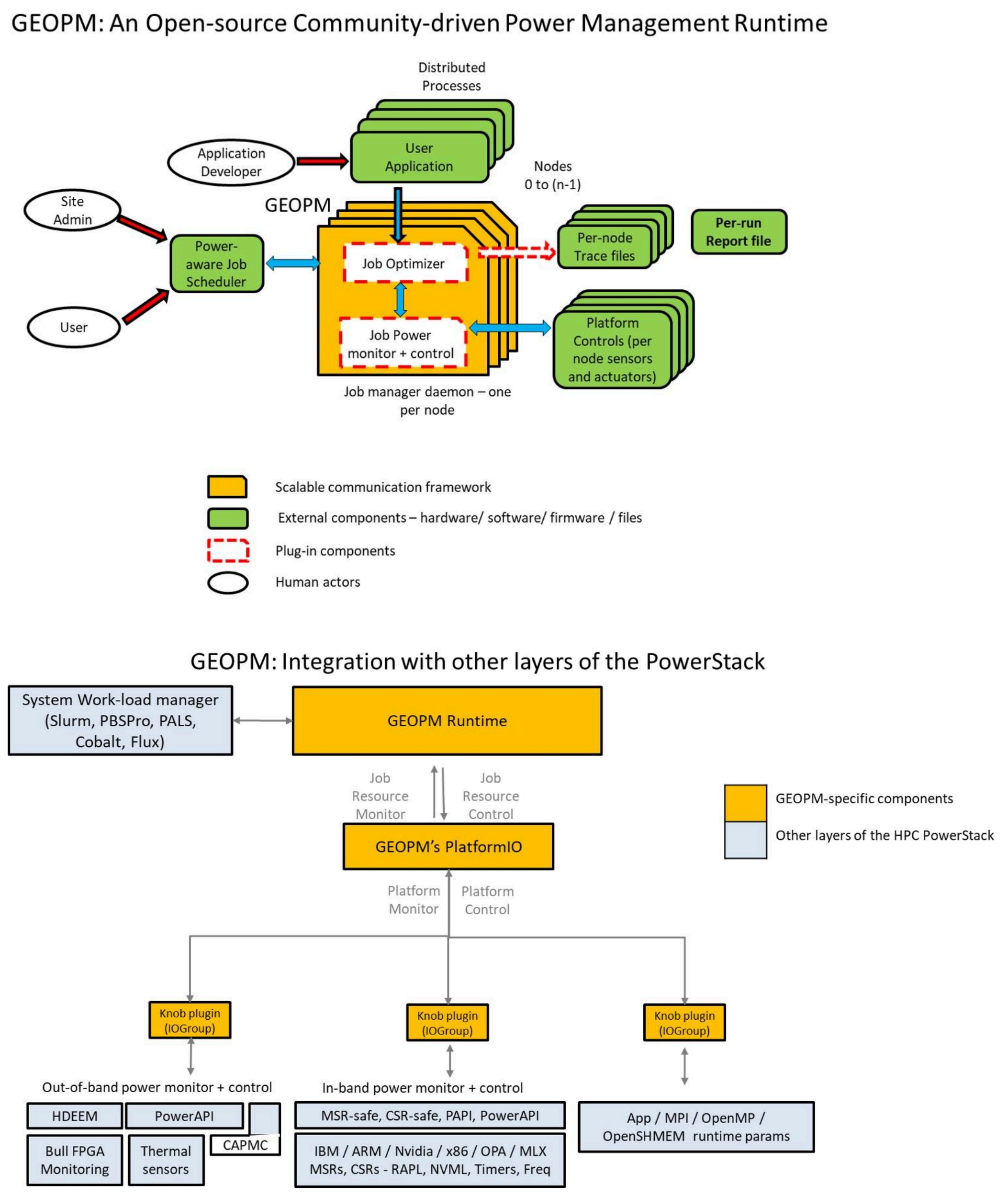
GEOPM: An Open-source Community-driven Power Management Runtime
Distributed Processes
Application Developer
User Application
Nodes 0 to (n-1)
Site Admin
GEOPM
Power-aware Job Scheduler
Job Optimizer
Per-node Trace files
Per-run Report file
User
Job Power monitor + control
Platform Controls (per node sensors and actuators)
Job manager daemon – one per node
Scalable communication framework
External components – hardware/ software/ firmware / files
Plug-in components
Human actors
GEOPM: Integration with other layers of the PowerStack
System Work-load manager (Slurm, PBSPro, PALS, Cobalt, Flux)
GEOPM Runtime
Job Resource Monitor
Job Resource Control
GEOPM-specific components
Other layers of the HPC PowerStack
GEOPM's PlatformIO
Platform Monitor
Platform Control
Knob plugin (IOGroup)
Knob plugin (IOGroup)
Knob plugin (IOGroup)
Out-of-band power monitor + control
In-band power monitor + control
HDEEM
PowerAPI
CAPMC
Bull FPGA Monitoring
Thermal sensors
MSR-safe, CSR-safe, PAPI, PowerAPI
IBM / ARM / Nvidia / x86 / OPA / MLX MSRs, CSRs - RAPL, NVML, Timers, Freq
App / MPI / OpenMP / OpenSHMEM runtime params

Figure 3.6 GEOPM overview

3.2.2 Variorum

(Contributor: Stephanie/Tapasya/Aniruddha)

ToDo: To be written.

3.2.3 End-to-End Autotuning Framework in PowerStack

(Contributor: Xingfu/Aniruddha/Ondřej)

There are several existing research approaches on auto-tuning presented by the experts in this working group. These efforts include system and hardware tuning, runtime system-application co-tuning, compiler-level parameter tuning, loop-level parameter tuning, deep learning-based hyperparameter tuning, etc.

A primary limitation of most - if not all - of these efforts, is that the autotuning efforts have been conducted in isolation, limited only to the individual layers of the PowerStack. To address this limitation, the primary goals of this working group are as follows:

a) Opportunity analysis: How do we quantify the potential benefits of end-to-end tuning across the layers of the PowerStack? What experimentation is required to achieve baseline quantification of the benefits of end-to-end tuning?

b) Based on the conceptual diagram of the PowerStack, how do we apply our current autotuning work currently limited to the individual layers of the PowerStack?

c) How do we combine the existing work by experts at different layers of the PowerStack?

In order to answer these questions, we took a survey of the expertise/background of participants of this working group. The input helps us guide the collaborative effort towards addressing difficult research problems in this domain. We identify the opportunities for collective autotuning of two or more management layers in the PowerStack. We find potential areas for research and prepare a list of broad research questions that we are collectively interested in tackling.

Framework, Methodology, and Opportunity

HPC PowerStack framework has four layers in a high-level view: Site, System, Jobs / Applications, and Nodes. A site has a power budget. System is constrained under the budget. Therefore, we focus on tuning at system, Jobs/Applications, and nodes and propose a high level overview of end-to-end autotuning framework in HPC PowerStack as shown in Figure 3.7.. We define tunable parameters at each layer, then discuss how to autotune the combination of different parameters at the distinct layers (parameter space) for an optimal solution (the smallest runtime, the lowest power, or the lowest energy) under a system power cap.

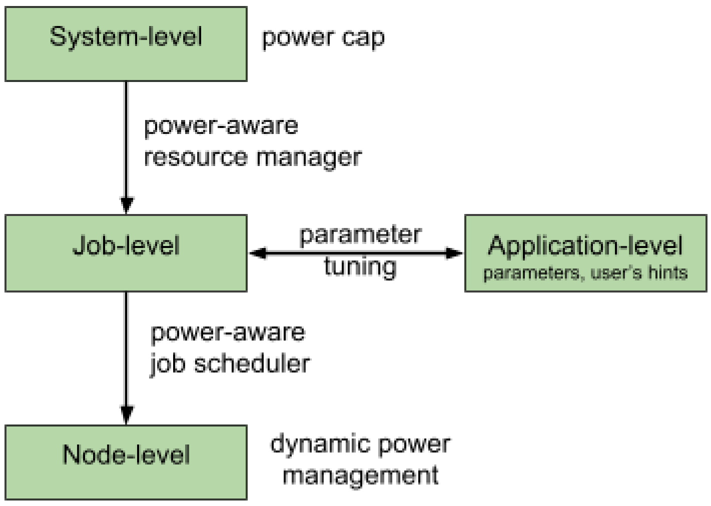

Figure 3.7 End-to-End Autotuning Framework (draft)

We describe the knobs at each level, who controls, what control knobs can be modified at what time (temporal) and space (spatial), what to measure. We also address the following questions: goal/objective, metrics, parameters, and methods for each layer and/or across two or more layers.

1) System level Tuning

The objective for this layer is power-constrained performance optimization, energy-constrained performance optimization, guarantee rate of change in system power usage (in a specified time window), lower and/or upper bounds on system power usage, and thermal-constrained performance optimization. The main metrics are system power, energy and utilization, ambient temperature, water temperature for liquid-cooled systems, job turnaround time, queuing delay, and job throughput. The main tuning parameters are number of nodes to bring up/allocate (hand over to RM for management), liquid flow for liquid-cooled systems, and policy definition (job power limit, number of nodes) to RM. The tuning methods are system level job management: canceling running jobs, pausing/restarting jobs, job placement; out-of-band power and/or energy controls; dynamic resource (power, energy) management, and power-aware job scheduler -- service level agreement (SLA) for the job.

2) Job level Tuning

The objective for this level is power/energy-constrained performance optimization, guarantee a lower bound on the performance of the job, lower bound on job-level power usage, best-effort performance optimization in compliance with RM SLA. The main metrics job-level power or energy usage, execution time, operating frequency, performance (FLOPS, IPC, IPS), power efficiency (FLOPS/watt, IPC/watt), and energy efficiency (ED, ED^2, FLOPS/Joule, IPC/Joule). The parameters are node level power limit, device level power limit (socket power limit, GPU power limit, DRAM limit, etc.), DVFS, clock modulation, concurrency (# of threads, # of processes) per compute device, idle cycle injection, and application-level parameters (algorithm choice, sub-algorithm options, input-specific options, etc.). The methods are energy/power shifting between compute devices, energy saving using frequency scaling (DVFS) strategy according application phases (I/O, memory-bound, communication-bound, compute bound), uncore frequency scaling for saving energy, and thread and process placement.

3) Application level Tuning

This level is related to how to tune application using compilers, program models such as MPI and OpenMP, CUDA, OpenCL, other software dependencies. The objective for this level is performance optimization and reducing memory footprint, reducing data movement and I/O footprint. The metrics are power, energy, runtime, and power efficiency. The parameters are DCT (#threads, #processes), DVFS, application-specific parameters: (algorithm choice, sub-algorithm options, input characteristics), system parameters: cache behavior, memory allocation and access (page size), etc., and JIT enable parameters. The methods are to tune runtime paradigm (CUDA, OpenMP, MPI, Charm++) for performance, malleability, elasticity in MPI, OpenMP-specific parameters: scheduling, block size, to tune compiler optimization parameters (externally) for performance, and to utilize the huge page sizes.

4) Node level Tuning

The objective is to apply node (or device) level power limit, and resource optimization and power balance. The metrics are power usage, energy usage, performance (FLOPS, IPC, IPS), power efficiency (FLOPS/watt, IPC/watt), energy efficiency (ED, ED^2, FLOPS/Joule, IPC/Joule), and node utilization. The parameters are node power limiting, node power assignment policy, core and uncore frequency scaling, out-of-band controls and in-band controls, and memory modes. The methods are dynamic power/energy management and shifting among individual compute/memory/I/O devices on the node, DVFS/power capping to reduce power usage of idle components, and to configure high bandwidth memory as cache to reduce memory power.

5) List of existing software components for this framework

We list several existing software components at the four layers for this framework. By integrating proper software components from each layer, we are able to do the proposed end-to-end autotuning in PowerStack.

(TODO: add citations for each of the components below)

Table 3.2 List of software components

PowerStack Layers	**Current state-of-the-art software components**
Resource Manager / Job Scheduler	SLURM, FLUX, PBS, PERQ, Cobalt, ALPS/PALS, and iRM (SLURM extension with Dynamic Resource Management capabilities)
Job-level runtime system	GEOPM, READEX, Conductor, Uncore power scavenger, and COUNTDOWN
Node-level management	PlatformIO, Variorum, Libmsr, PowerAPI, x86_adapt, and Cpufreq, NVML, nvidia-smi
Applications-level tuning	application parameter tuning, compiler- and loop-level autotuning Y-TUNE (ytopt), JIT (Just In Time) compilation, different implementations of an application

6) Research Opportunity

We identify the opportunities for collective autotuning of two or more management layers in the PowerStack. The goal is to find potential areas for research and prepare a list of broad research questions that we are collectively interested in tackling. For co-tuning resource manager and runtime system, the objective is to define interactions (static and dynamic) based on the type of job (moldable/malleable vs non-moldable/malleable) and job-agnostic interactions that are common to the above two types of jobs. For Co-tuning resource manager and application + software, the objective is to target how the application fully utilizes the allocated resources for efficient execution under the job power cap. For co-tuning runtime system and application + software, the objective is to fine-tune system parameters, software and application parameters at run-time under job power cap. For co-tuning resource manager, runtime system and application + software, the objective is about how the applications fully utilizes the dynamically allocated resources for efficient execution with the maximum job throughput under the system power cap. Overall, when we consider tuning all four layers, we have to carefully analyze the tunable parameters from each layer to identify how they impact each other across layers to find the best combination for the optimal performance under a power budget.

We discuss some challenges and open issues. Collective tuning of powerstack layers has remained to be a challenging task due to lack of interfaces for (1) translation of high-level goals into subsequent lower level goals, (2) translation of monitored metrics at lower layers to derived metrics at higher layers, and (3) enabling custom configuration (e.g. resource reallocation and remapping) at the launch of during job runtime. Developing such interfaces will provide the platform to explore several research directions for the reallocation of resources to job and sub-job components.

Several research challenges exist in fine-tuning control parameters exposed by different software components in the PowerStack. The key motivation behind such a tuning is that individual layers in the stack are tuned based on local view of those layers without taking into account the impact of the tuning decisions outside of those layers. There are several interesting research questions that can be explored in this space such as how do the existing control parameters exposed by the different layers of the powerstack and the application interact with each other on the scale of power efficiency? How to find the correlation between the characteristics of the allocation and management algorithms used by the layers of the powerstack and how those algorithms interact with the application? Can we extend the algorithms at different layers of the powerstack to incorporate semantic information in the application?

3.2.4 Power aware scheduling

(Contributor: TBD)

ToDo: To be written.

3.2.5 Argo/NRM (Contributor: Valentin/Swann)

The Argo ECP project is building portable, open source system software that improves the performance and scalability and provides increased functionality to Exascale applications and runtime systems. We focus on four areas of the OS/R stack where the need from the ECP applications and facilities is perceived to be the most urgent: 1) support for hierarchical memory; 2) dynamic and hierarchical power management to meet performance targets; 3) containers for managing resources within a node; and 4) internode interfaces for collectively managing resources across groups of nodes.

Our main software effort within the second category is the Node Resource Manager(NRM), an effort towards reconfigurable node-local resource management. The goals of the NRM effort are dual:

First, provide a unified gateway for global resource managers to change the way node resources are utilized. In summary, the NRM API exposes:

- Wrapped execution calls (start/kill jobs/containers)
- Sensors (application performance information, power utilization, CPU counters)
- Actuators (Power settings, including Variorum passthrough)

Second, provide an autonomic node-local control strategy to manage these resources according to node-level Goals and Constraints.

The NRM infrastructure consists of a client-daemon system, shown in Figure 3.8.

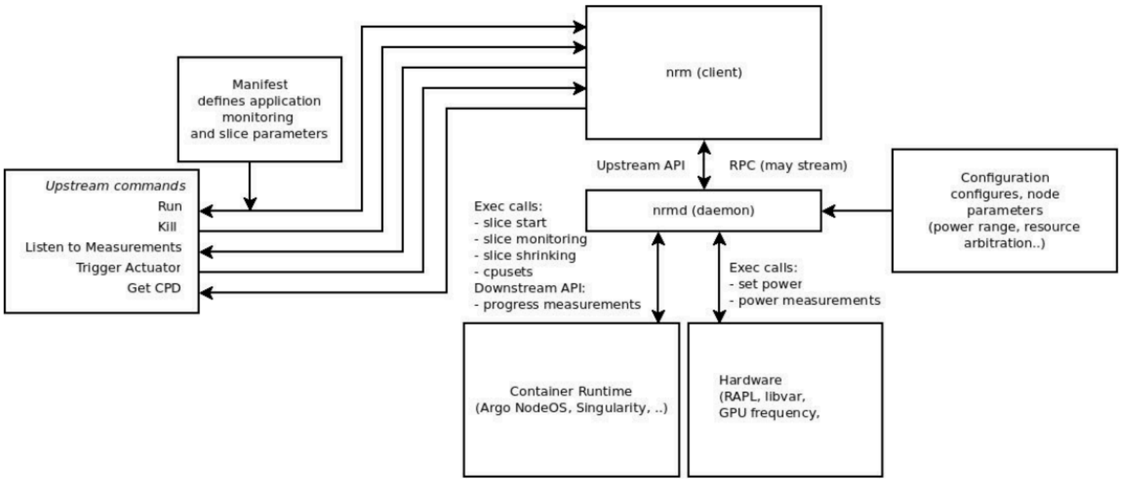

Figure 3.8 NRM infrastructure

While assumed to be rather stable throughout the execution of a computational job, the precise list of Sensors, Actuators, Goals and Constraints is in effect subject to arbitrary change at runtime. For instance, an user-level process might decide to start an application on the node, which would create a new application performance sensor, set a goal or constraint, or enable/disable a power control mechanism, which would create a new actuator. This requirement of runtime reconfigurability of the exposed interface (and ergo of the corresponding resource management strategy) underlines a need for a systematic approach to the management of the meta-data associated with what we've described as Sensors, Actuators, Goals and Constraints. In order to clear the path towards a systematic management of this meta-data, the NRM effort includes the development of a Control Problem Description (CPD) format, whose purpose is to serve as a bridge between a compute node and the resource management strategy, be it autonomic (through NRM itself) or deferred to a global resource manager. The following figure illustrates the frontier between NRM, the Control Problem Description, and the node resource manager.

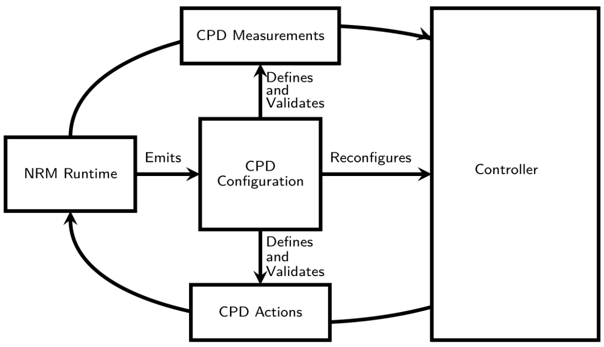

Figure 3.8 NRM infrastructure

As a way of example (and rather than describing the complete schema of permissible CPD values), we offer the following value for a CPD generated at runtime by NRM. This CPD has been generated for a one RAPL-enabled package node, with one application running, whose performance is estimated from CPU counters. The node-local Goal has been deemed the minimization of performance-adjusted power expenditure, and a constraint on the reduction in average application performance has been set. The single actuator has been configured to allow two settings (low powercap, 100 Watts, and high powercap, 200 Watts).

```
Actuators:
- RaplKey (PackageID 0) : Actuator {actions = [DiscreteDouble 100.0,DiscreteDouble 200.0]}
Sensors:
- ProgressSensorKey (CmdID c3de4[..]) : Sensor {range = 0.0 ... 100000.0, maxFrequency = 1.0}
- RaplKey (PackageID 0) : Sensor {range = 0.0 ... 150.0, maxFrequency = 3.0}
Objectives :
-weight 1.0: ("RaplKey (PackageID 0)"+0.0+200.0)/(max(0.5,min(1.5,("ProgressSensorKey (CmdID c3de4[..])"+0.0)/("REF:ProgressSensorKey (CmdID c3de4[..]")"+0.0+1.0))))
Constraints:
-threshold 0.9: max(0.5,min(1.5,("ProgressSensorKey (CmdID c3de4[..])"+0.0)/("REF:ProgressSensorKey (CmdID c3de4[..])"+0.0+1.0))
```

The CPD format exposes a dictionary of precise meta-data about the sensors and actuators it exposes. In turn, Objectives and Constraints are described in terms of simple abstract syntax trees over sensors keys. The design of the CPD objective AST allows for the evaluation of constraints and objectives, as well as the evaluation of the upper and lower bounds on their values, which together constitute a valuable asset for reconfigurable control.

Just as NRM exposes the current “control problem” to an upstream controller at any time, it also provides various control algorithms, mostly variations of so-called adversarial multi-armed bandit strategies.

6. Conclusion

(Contributor: Masaaki)

Power and energy consumption are now the first class design and operating constraint of HPC systems. The HPC community needs a holistic stack for power and energy management. In this white paper, we provide a high-level overview of the HPC PowerStack, a software stack which manages power and energy of HPC systems. We expect significant update of the PowerStack strawman design and use cases over time, and we expect the design and refinements to be a collaborative effort across a broad community.

Appendix A. Glossary - definition of terms

(Contributor: everyone)

- Node - a node is a complete and functional hardware entity that can run a process(es). A node may contain the power supply and the point of instrumentation for collecting power or energy samples for a node or multiple nodes (e.g., Chassis design).
- System - a system is a basic, complete, and functional hardware and software setup with everything needed to implement HPC computing performance.
- Job - a job generally refers to a distributed or task-parallel job in the PowerStack. A job is running multiple tasks independently or jointly across multiple processors (or cores/threads).

Appendix B. Glossary of Time-scale

(Contributor: everyone)

This section shows typical execution time / time bounds of different HPC system entities. Note that the ranges of the time-scale may change as this document is updated.

Make an explicit disclaimer mentioning that the time-scale ranges may change as this document is updated over time.

- Application runtime: The average application phase time over which a knob controls make sense.
  - ToDo: Example of time-scale

- System level runtime: The granularity at which system-level power-caps need to operate.
  - ToDo: Example of time-scale

References

[1] “PowerStack: A global response to the power management problem for exascale”, HIPEAC 2019 Blog Article

[2] “A Strawman for an HPC PowerStack”, OSTI Technical Report, August 2018

[3] "Energy and Power Aware Job Scheduling and Resource Management: Global Survey — Initial Analysis," M. Maiterth et al., 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Vancouver, BC, 2018, pp. 685-693, doi: 10.1109/IPDPSW.2018.00111.

[4] J. D. Ullman, “NP-complete scheduling problems”, Journal of Computer and System sciences, 1975

[5] R. E. Grant, M. Levenhagen, S. L. Olivier, D. DeBonis, K. T. Pedretti and J. H. Laros III, "Standardizing Power Monitoring and Control at Exascale", in Computer, vol. 49, no. 10, pp. 38-46, Oct. 2016. doi: 10.1109/MC.2016.308 BibTeX.

[6] R.E. Grant, B. Rountree, J. Hansen, et al., “High Performance Computing Power Application Programming Interface Specification Community Version 1.0”, Technical report, November 2019, Available at: https://github.com/pwrapi/powerapi_spec/releases/

[7] https://eehpcwg.llnl.gov/

[8] Distributed Management Task Force, Inc.(DMTF), “Redfish White Paper”, (https://www.dmtf.org/sites/default/files/standards/documents/DSP2044_1.0.4.pdf).

[9] Intelligent Platform Management Interface (IPMI), “IPMI Technical Resources”, https://www.intel.com/content/www/us/en/servers/ipmi/ipmi-technical-resources.html

[10] The HPC PowerStack. https://powerstack.caps.in.tum.de/