A Scheduler-Level Incentive Mechanism for Energy Efficiency in HPC

Yiannis Georgiou
BULL
Yiannis.Georgiou@bull.net

David Glesser
BULL
and Univ. Grenoble-Alpes
David.Glesser@bull.net

Krzysztof Rzadca
Institute of Informatics
University of Warsaw
krz@mimuw.edu.pl

Denis Trystram
Univ. Grenoble-Alpes
and Institut Universitaire de France
trystram@imag.fr

***Abstract*—Energy consumption has become one of the most important factors in High Performance Computing platforms. However, while there are various algorithmic and programming techniques to save energy, a user has currently no incentive to employ them, as they might result in worse performance. We propose to manage the energy budget of a supercomputer through EnergyFairShare (EFS), a FairShare-like scheduling algorithm. FairShare is a classic scheduling rule that prioritizes jobs belonging to users who were assigned small amount of CPU-second in the past. Similarly, EFS keeps track of users' consumption of Watt-seconds and prioritizes those whom jobs consumed less energy. Therefore, EFS incentives users to optimize their code for energy efficiency. Having higher priority, jobs have smaller queuing times and, thus, smaller turn-around time.**

To validate this principle, we implemented EFS in a scheduling simulator and processed workloads from various HPC centers. The results show that, by reducing it energy consumption, a user will reduce it stretch (slowdown), compared to increasing it energy consumption. To validate the general feasibility of our approach, we also implemented EFS as an extension for SLURM, a popular HPC resource and job management system. We validated our plugin both by emulating a large scale platform, and by experiments upon a real cluster with monitored energy consumption. We observed smaller waiting times for energy efficient users.

***Index Terms*—Resource and Job Management Systems, Scheduling, FairShare, Energy-Aware, Energy-Efficiency**

I. Introduction

A modern large-scale supercomputer runs on huge amounts of electrical power. For instance, Tianhe-2 (the leading system of the TOP500) develops 33 Petaflop/s and consumes a power of 18 MWatts [1]. About half of the cost of a Petascale system comes from energy consumption, and today, it costs about 1 million dollars a year to run a 1 MWatts system. This means that the electricity bill is roughly equal to the hardware cost of such platforms. However, the cost of the electricity is not the sole problem of high energy consumption. As the systems are more and more densely integrated, the heat generated as a direct result of energy consumption is more and more difficult to disseminate. The energy consumption is the most important obstruction for building exascale machines [2].

Various methods have been proposed for reducing energy consumption. The design of an energy-efficient supercomputer involves for instance accelerators (GPUs or Intel Phis), which are more energy-efficient than standard CPUs for some regular workloads; or, water cooling, which disseminates heat more efficiently than air-conditioning. Yet, once a system is built, such "black-box" approaches have limited effects. For instance, by reducing the speed of processors, we reduce their voltage, and thus their power consumption. However, not only the speed-scaling slows down the computations; more importantly, the total savings are very limited. In our previous research [3], we speed-scaled processors of a cluster while measuring run-time and energy consumption of typical HPC applications, from computationally-intensive LinPack to communication-bounded Intel MPI benchmarks. Depending on the type of applications, speed-scaling increased the runtime twice; yet, electricity used was lowered by only up to 30%. Thus, we need to motivate the user to actively participate in making her computations energy efficient.

Significant savings can be made by designing energy-efficient software. The methods range from changing algorithms (e.g., by reducing communication [4]), to requesting a lower processor voltage when the application enters a non-computationally-intensive phase. The key to these savings is that only the programmer is able to precisely decide when to slow down the hardware without a large impact on the code's observed performance.

However, in a supercomputer shared by many users, there is no incentive for users to adapt energy-efficient software—and usual, minor deterrents from doing so, ranging from anxiety about performance to a "If it ain't broke, don't fix it" attitude.

We propose to shift the focus of a scheduling policy from processors to what is currently the true limit of large scale supercomputers: energy. We assess this idea by creating EnergyFairShare (EFS) scheduling algorithm. EnergyFairShare uses a well-known algorithm for sharing resources, FairShare; but the resource that is to be shared fairly is the energy budget of a supercomputer; not its processors. Consequently, users' jobs are prioritized according to their past energy consumption. Once a particular user exceeds her assigned share of the total energy, the priority of her jobs is lowered; thus, in a loaded system, the jobs stay longer in the queue. This mechanism creates an incentive for users to save energy.

Moreover, EnergyFairShare may be used to achieve fairness in large supercomputing centers managing heterogeneous resources—from various kinds of cluster nodes (e.g., fat nodes, accelerators, FPGAs) to machines (e.g., an x86 cluster, or a

BlueGene) to specialized equipment. Each resource type can be abstracted and characterized by its energy consumption. In a supercomputing center managing heterogeneous resources under a common energy budget, a user would have an incentive to choose the most energy-efficient resource for her needs.

We tested EnergyFairShare by simulation to assess the impact of jobs' energy efficiency on their queuing performance. As there are no available workloads that show jobs' energy consumption, we extended the standard workloads by assigning to each job its simulated consumption which was based on the job's size. Our results show that the energy efficient jobs have, on the average, a stretch (slowdown) lower than the stretch of the standard and gluttonous jobs.

To validate our ideas in real-world settings, we implemented EnergyFairShare as a scheduling plugin for SLURM [5], a popular HPC resource and job management system (RJMS). Our plugin is compatible with existing FairShare policies. It obtains jobs' energy consumption data through SLURM energy accounting framework. EFS rewards energy-efficient executions with higher prioritization within the scheduler.

The remainder of this paper is organized as follows. Section II presents related work, Section III provides the description of the EnergyFairShare algorithm along with the implementation details. In Section IV, we validate our algorithm and implementation through several experimentations. Finally, we conclude and discuss ongoing works in Section V.

II. Related work

The main contribution is to propose to treat the energy as the limiting resource and to share the available energy budget in a fair way using the classical FairShare algorithm. Thus, below we argue that (1) in existing HPC resources, it is possible to account to a job the energy it consumes; (2) users can save energy by choosing energy-aware algorithms and libraries (thus, it makes sense to provide an incentive to save energy); (3) FairShare is a standard algorithm to achieve CPU fairness between users (many alternative approaches and algorithms exist, but some of them can be adapted to energy similarly to our approach).

A. Measuring energy consumption

In order to manage the energy budget of a supercomputer, we need to precisely measure the total energy consumed by each job. Fortunately, energy consumption is, or quickly becomes, a key issue in very diverse types of resources: from mobile devices, in which the goal is to extend the battery lifetime, to supercomputers. Thus, modern hardware provides various interfaces to monitoring energy consumption. Standards include the Intelligent Platform Management Interface (IPMI, [6]) that uses a specialized controller (Baseboard Management Controller, BMC) to monitor various hardware parameters of a system, including power. Intel RAPL (Running Average Power Limit, [7]) enables the operating system to access the energy consumption information for each CPU socket. This information is computed from a software model driven by hardware counters. It is also possible to measure the GPU power consumption [8]. Energy consumption of CPU and each different component can be also approximated through models [9], [10]. However, not all hardware components (CPU, RAM, networks cards, switches, etc.) are equipped with built-in sensors. Alternatively, external power meters can monitor whole nodes (and also other equipment like switches or routers): for instance, [11] describes a deployment of power meters on three clusters; [12] proposes a software framework that integrates power meters in datacenters. While an external power meter should be more precise in measuring the total consumption (as the information measured is closer to what the electric utility measures through their electric meters), when a few jobs share a single node, it is not clear how each job should be accounted for the usage. Hackenberg et al. in [13] introduce a high definition energy efficiency monitoring infrastructure that focuses on the correctness of power measurements and derived energy consumption calculations. This infrastructure is based upon temporal resolution optimizations through internal BMC polling and querying via IPMI. They demonstrate improved accuracy, scalability and low overhead with no usage of external wattmeters. They also describe the architecture of new FPGA based measurement infrastructure with spatial granularity down to CPU and DRAM, temporal granularity up to 1000 sample/s and accuracy target of 2% for both power and energy consumptions. SLURM has recently been enhanced by introducing the capability to regularly capture the instantaneous consumed power of nodes [3]. Based upon this power-aware framework, SLURM is the RJMS to provide energy accounting and power profiling per job. However, we argue that since power measurements take place only at the node level, the derived energy calculation will not reflect reality if jobs make use of nodes' parts or if they share nodes with other jobs.

B. Saving energy in HPC

Currently, there are two main approaches to improve energy efficiency in HPC: static power management, or designing hardware operating on efficient energy levels (e.g. low voltage CPUs used in IBM BlueGenes); and dynamic power management in which software dynamically switches the voltage and frequency (DVFS) used by a component [14]. Designing efficient hardware is orthogonal to the scope of this paper as we assume that an HPC platform is given; thus below we review the dynamic approaches. The core idea is to lower the frequency of the processor (which lowers the power consumption) when the job enters a computationally-light phase (recent surveys are [14], [15]). DVFS can be used by the cluster scheduler without knowing the workload of an application (the frequency is dynamically adapted to the load on each processor); however, the energy savings of such black-box approaches are limited (a review [14] reports energy savings on NAS Parallel Benchmark of up to 20-25% with roughly 3-5% performance degradation).

Other methods complement DVFS. A simple, system-level technique is to switch off unused processors—here the key problem is the energy needed to switch the processor back

on [16]. The savings increase when a whole node, or a whole rack is switched off.

Further savings require adapting the application. For instance, in a distributed application that cannot be perfectly load-balanced, processors assigned smaller loads can be slowed down (by DVFS) to finish in roughly the same time as more loaded processors [17]. In distributed algorithms, communication between nodes is expensive in terms of energy; communication-optimal [4] or communication-avoiding algorithms reduce communication (sometimes increasing the amount of per-core computation).

To summarize, apart from black-box DVFS (which results in limited gains) and switching off the idle resources (of limited use, since most modern supercomputers are constantly overloaded), the approaches require the programmer to instrument the code, or even to change the algorithms. Moreover, saving energy incurs performance loss. Thus, a user must have incentives for saving energy. EnergyFairShare gives higher priority, thus lower queuing times, for users with smaller energy needs.

C. Fairness in HPC resource management

Fairness, even restricted to HPC, is a vast research area. The most popular approach is the *max-min fairness* [18], in which the goal is to maximize the performance of the worst-off user. Production schedulers like LSF [19], Maui/Moab [20], TORQUE [1] or SLURM [5], use this approach through the FairShare algorithm. Usually the scheduler accounts for each CPU-second used by each user, decayed over time. Users with small total CPU-second usage have priority over users with large usage. Fair-share is compatible with the typical workflow of a scheduler (assign priorities, sort jobs, schedule according to priorities); moreover, the priorities can be further modified by static site policies (e.g., weighting groups of users in function of their payments to the site). EnergyFairShare uses FairShare, but the users are accounted for joules (watt-seconds), instead of CPU-seconds.

Many alternatives to FairShare were proposed; here we just list a few recently proposed. Klusaček et al. [21] modify conservative backfilling to improve fairness. Emeras et al. [22] proposes an algorithm optimizing the slowdown (the stretch) of each user's workload. The algorithm uses the concept of a virtual schedule, in which CPUs are assigned to users' workloads; thus it can be modified to treat the energy as the primary resource in a similar way as we modify FairShare.

EnergyFairShare can manage heterogeneous resources, as each resource can be characterized by its energy needs. Klusaček et al. [23] reviews multi-resource fairness. Papers differ by their definition of what a multi-resource fair schedule is [24], and more specifically on the properties that their algorithm guarantee. The Dominant Resource Fairness algorithm proposed by Ghodsi et al. [18] guarantees sharing incentive, strategy proofness, envy freeness and Pareto efficiency. The algorithm proposed by Klusaček et al. [23] guarantees multi-resource awareness, heterogeneity awareness, insensitivity to scheduler decisions, walltime normalization, support for multi-node jobs. TORQUE and Maui/Moab support multi-resource fairness by counting resource usage using a measure different from CPU-seconds to count resource usage. TORQUE computes distance to a standard (mean) job. Maui/Moab [20] computes Processor-Equivalents, transforming consumption of non-standard resources to normalized CPU-seconds.

To our best knowledge, there is no other work proposing fair resource sharing based on energy. In the following section we argue that a energy fairsharing implies a multi-resource fairsharing.

[1] http://www.adaptivecomputing.com/products/open-source/torque/

III. EnergyFairShare Algorithm

EFS modifies FairShare to consider energy instead of CPUs as a main resource. Therefore, we start by describing the environment in which EFS works (a resource management system); then we describe the classic FairShare algorithm; and finally—the principle and the implementation of our algorithm. To better ground our discussion, next to discussing ideas behind these, we will show how they are realized in SLURM (the resource manager in which we implemented EFS).

A. Scheduling in Resource and Job Management Systems

Scheduling in a standard RJMS (such as SLURM, Maui, etc.) consists of two successive phases. First, pending jobs are prioritized according to some criteria. Then, picking jobs one by one in order of the assigned priorities, the scheduler assigns resources to jobs. EnergyFairShare is a prioritization algorithm; thus existing, efficient algorithms (such as backfilling [25]) may be used in the second phase.

The priorities computed in the first phase are usually a linear combination of factors based on various parameters of a job. Example factors include job's waiting time (the longer the job queues, the higher is its priority); job's size (priority of long jobs may be reduced to increase job throughput); job's owner (to prefer accounts associated with a project that funded the supercomputer). Fair-share, described in the next section, may be used as an additional factor. Usually, a weight is assigned to each of the above factors. Weights allow to enact a policy that blends a combination of any of the above factors in any desired portion. For example, a site could configure FairShare to be the dominant factor while setting job size and age factors to each contribute a smaller part.

B. Computing Priorities by Fair-Share

The FairShare algorithm computes queued jobs' factor (priority) based on the amount of resources consumed by the job's owner in the past. The job's FairShare factor is commonly added to other factors described in the previous section. The FairShare factor serves to prioritize queued jobs such that those jobs charging accounts that are under-serviced are scheduled first, while jobs charging accounts that are over-serviced are scheduled when the machine would otherwise go idle. Also, FairShare generalizes to hierarchies of accounts so that not only the owner's usage, but her group's, or her supergroup's, is taken into account (here, for simplicity of presentation we will

not discuss it; we assume that each user has a single account and that the scheduling policy is based on these accounts).

Basically there are two parameters that influence SLURM FairShare factor: i) the normalized shares as defined in the associations of the database; and ii) the normalized usage of computing resources as a continuously evolving parameter computed from the accounting database.

A scheduling policy defines a target share s_u of a system for each account u. The algorithm computes, for each account u, the normalized share S_u as

$$S_u = s_u/(\sum_v s_v). \quad (1)$$

S_u expresses the share of the system that, on the average, user u is entitled to use.

The usage is computed as a total amount of consumed resources normalized by the amount of available resources; usually, recent usage counts more than the past (the usage is decayed over time). To compute the normalized usage, once every job completes, a RJMS stores in an accounting database the job's runtime multiplied by the amount of CPUs assigned (CPUs, as historically this was the most contested resource). The raw usage R_u for user u is computed based on a half-life formula that favors the most recent usage data. Past usage data decreases based on a single decay factor, D:

$$R_u = R_u(\delta_0) + D \cdot R_u(\delta_1) + D^2 \cdot R_u(\delta_2) + \ldots, \quad (2)$$

where δ_i is the ith measurement period counting from the current time moment (e.g.: δ_0 is the last 24 hours; δ_1 is the previous 24 hours etc.); and $R_u(\delta_i)$ is the number of CPU-seconds used by u during period δ_i. To get the normalized usage U_u, R_u is normalized by the total amount of available resources decayed over time,

$$U_u = R_u/(\delta_0 \cdot m + D \cdot \delta_1 \cdot m + D^2 \cdot \delta_2 \cdot m \ldots), \quad (3)$$

where m is the number of available CPUs. For instance, assume that on a $m = 50$ CPU system only the last 100 hours are taken into account ($\delta_0 = 100 \cdot 3600$, $D = 0$). If, during this period, a user completed 5 jobs (30-hour long) each taking 10 CPUs, her normalized usage is $(5 \cdot 30 \cdot 10)/(50 \cdot 100)$, or 0.3.

Various functions are used to convert share and usage to a priority value. For instance, SLURM computes the FairShare factor F_u for a user u as:

$$F_u = 2^{-U_u/S_u/d}, \quad (4)$$

where d is an additional damping parameter. Consequently: a user with no usage gets FairShare factor of 1; a user with usage equal to her share gets the factor of 0.5; and a user whose usage vastly exceeds her share gets the factor close to 0.

C. EnergyFairShare: the principle

EnergyFairShare, the algorithm we propose, uses the FairShare algorithm described above, but counts Joules (energy over time, Watts per second) instead of CPU-seconds. Thus, the accounting module keeps track of the energy consumed by each job, which requires it to get the data from the cluster's energy monitoring system. Then, the job's EFS priority is computed according to the owner's energy usage and its assigned share of the total energy budget (just as in the FairShare algorithm). The resulting value may be treated just as the FairShare priority is, so it may be added to other factors (job age, size, or even classic, CPU FairShare), to get the final priority.

D. EnergyFairShare as a SLURM scheduling feature

We implemented EFS upon the open-source resource and job management system SLURM [5]. As of the June 2014 Top500 supercomputer list [2], SLURM was performing workload management on six of the ten most powerful computers in the world including the number 1 system, Tianhe-2 with 3,120,000 computing cores.

SLURM is designed as a client-server distributed application: a centralized server daemon, also known as the controller, communicates with a client daemon running on each computing node. Users can request the controller for resources to execute interactive or batch applications, referred as jobs. The controller dispatches the jobs on the available resources, whether full nodes or partial nodes, according to a configurable set of rules.

The SLURM controller also has a modular architecture composed of plugins responsible for different actions and tasks such as: job prioritization, resources selection, task placement, or accounting. We modified two of these plugins: the accounting plugin, to gather energy usage data; and the job prioritization plugin, where EFS is implemented.

SLURM has a particular plugin dedicated to gather information about the usage of various resources per node during job execution. This plugin, which is called *jobacct_gather*, collects information such as memory or CPU utilization of all tasks on each node. Then, the values are aggregated across all the nodes on which a job is running; then, two values per job are returned: the maximum and the average. These values can be then used for accounting, monitoring or scheduling. We extended this plugin to collect information from energy consumption sensors (however, we recall that measuring energy has its limitations, see Section II-A).

Various plugins can be used for job prioritization; we implemented EFS as an extension to the *multifactor* plugin, since the plugin uses the prioritization framework described in Section III-A and since it implements the FairShare algorithm. For a job, the result of EFS is treated as a factor, and added to other factors for a job, such as age or size.

IV. Experiments

We performed three different kinds of experiments with EFS. First, we implemented EFS in a simulator to run various traces and to check how changing jobs' energy efficiency influences their stretch. Second, we tested the EFS implemented as a SLURM extension: we emulated a particular supercomputer

[2] http://www.top500.org/list/2014/06/

Trace	Config	efficiency	Stretch normalized by baseline				
			Min	Mean	Med	Std Dev.	Max
SDSC	fs	both	1.00	1.00	1.00	0.00	1.00
SDSC	efs	green	0.27	0.91	0.92	0.33	1.82
SDSC	efs	gluttonous	0.27	1.10	1.02	0.31	1.66
SDSC	fs+efs	green	0.27	0.89	0.95	0.24	1.17
SDSC	fs+efs	gluttonous	0.27	1.03	1.01	0.34	1.92
PIK	fs	both	1.00	1.00	1.00	0.00	1.00
PIK	efs	green	0.73	0.97	1.00	0.07	1.00
PIK	efs	gluttonous	0.97	1.03	1.00	0.11	1.47
PIK	fs+efs	green	0.89	1.00	1.00	0.03	1.00
PIK	fs+efs	gluttonous	1.00	1.03	1.00	0.10	1.45
Curie	fs	both	1.00	1.00	1.00	0.00	1.00
Curie	efs	green	0.14	1.08	1.00	0.64	3.44
Curie	efs	gluttonous	0.33	3.02	1.02	6.25	25.7
Curie	fs+efs	green	0.14	1.63	1.00	2.45	13.5
Curie	fs+efs	gluttonous	0.17	1.57	1.00	2.06	11.2

TABLE I: Results of the simulations. In each simulation, one of the 20 most active users is either 30% more energy-efficient (green) or 30% less energy-efficient (gluttonous) than the rest. We compute the stretch for each job, normalized by the stretch in the baseline scenario; then average it over all jobs belonging to the same user. Rows presents statistics over 20 users.

(Curie, a 80640-core machine), and added to its trace users with varying (simulated) energy efficiency. Third, we tested the whole approach—from collecting energy usage to making scheduling decisions—by running the EFS-SLURM extension on a real, albeit small-scale, cluster.

A. Algorithm validation through simulations

We implemented EFS in Pyss [26], a discrete event simulator of batch schedulers. Once jobs are prioritized, the simulator uses the EASY backfilling to allocate resources. Waiting jobs are prioritized by three different policies:

- FairShare (FS): jobs owned by the user with the smallest recent CPU-second usage are prioritized (Section III-B). We set equal target shares S_u and the decay factor D to one week, as it is the default value for SLURM.
- EnergyFairShare (EFS): jobs owned by the user with the smallest recent energy (Watt-seconds) usage are prioritized (Section III-D). We set equal target shares and the same decay factor as FS.
- FS+EFS: FairShare and EnergyFairShare are normalized to their maximum current values and then summed.

All algorithms use job's arrival order to break ties.

We simulated three traces from the Parallel Workloads Archive [3]. We selected traces with high average usage to stress the scheduler (as in a lightly-loaded system, almost all jobs can be started immediately). Traces span different scales of HPC systems. The Curie trace is a 6 months trace of a 80640-cores machine ranked 26th on the top500 list of June 2014. The PIK trace is a 40 months trace of a 2560-cores cluster. Finally, the SDSC SP2 trace is a 24 months trace of a 128-cores cluster. We use the cleaned version of each trace. Results are presented in Table I.

[3]http://www.cs.huji.ac.il/labs/parallel/workload/logs.html

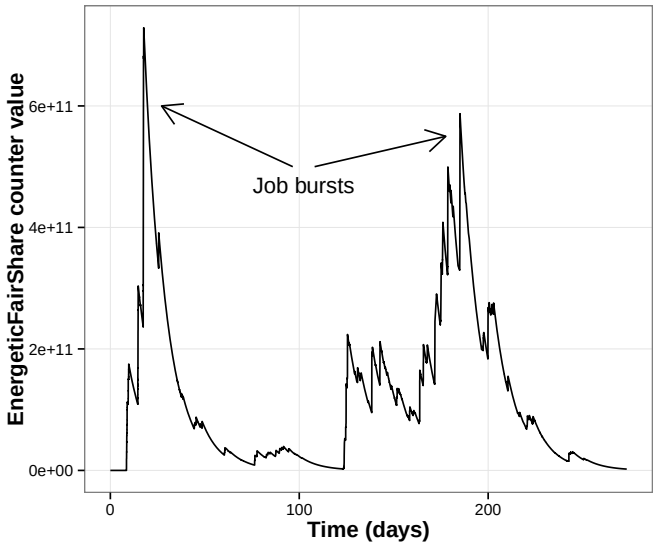


Fig. 1: Evolution over time of EnergyFairShare counter for user 33 during the experiment simulating Curie with the EFS policy. This user, even being 30% more energy-efficient than other users, worsens her mean stretch by 344%.

As it is possible to measure the nodes' energy consumption only relatively recently, the traces do not have the information about the energy consumption of jobs. In each experiment, we set the energy consumption to be proportional to the job's CPU-seconds. To get the baseline result, we first simulate each trace with each policy, where all job have the same energy efficiency. Then, we select the 20 users that submit the biggest number of jobs. For each selected user, we improve their jobs' energy efficiency by 30% (thus, a conservative estimate, since black-box DVFS report a 20-25% gain, Section II-B) and run the whole simulation (keeping other users not modified). Then, for each job of this user, we compare its stretch with the stretch in the baseline result by computing the factor `stretch-green / stretch-baseline`. Finally, to get an influence of the policy on the user, we compute the average of these factors (for all users' jobs). In Table I, the "green" rows show statistics (the minimum, the average, the median, the standard deviation and the maximum) of average factor on the sample of 20 users. Similarly, the "gluttonous" rows show statistics when the efficiency is *worsened* by 30% for each job.

As we expected, EFS (slightly) improves stretches of energy-efficient users. The difference is most visible on the SDSC trace, in which EFS reduces the mean stretch by 9% for the green users; and increases by 10% for the inefficient users. On the PIK trace, EFS reduces the mean stretch by 3% for the green users; and increases by 3% for the inefficient users. However, in the Curie trace, the results are less clear: although EFS distinguishes between green and gluttonous users (gluttonous users have, on the average, their stretch increased 3 times), green users are apparently 8% worse-off than in the baseline.

The worst-off "green" user has her stretch increased by 344%. We studied this user in detail. Figure 1 presents the evolution of the internal EnergyFairShare usage counter (U_u) through time. We see two peaks in the usage, corresponding to huge amounts of energy consumed by the users' jobs (we annotate these moments as job bursts). This implies that even with better energy efficiency, the user will have a high energy penalty at this moment of the trace. To be launched by the system, her jobs will have to wait until the usage decreases. We observed similar effects for other users.

Another effect that influences the Curie results is that the trace is very volatile—the standard deviations are, approximately, an order of magnitude greater than in SDSC and PIK. In this large system a small change in the scheduling decision can lead to a totally different schedule. For example, let us imagine a queue with a small and a huge job (the trace has quite a few jobs using 64.000 cores for more than an hour). If the scheduler chooses the huge job to be launched first, many other jobs will be delayed. Whereas if the small one is choosen first, thanks to backfilling, more jobs will be able to run before the huge job.

B. A Real implementation on an emulated platform

In this subsection, we test the implementation of our algorithm as a SLURM extension by emulating a large-scale supercomputer, Curie (a 5040 nodes, 80640 cores machine). Our experimental platform enables us to emulate 5040 nodes on 20 physical nodes by running multiple `slurmd` daemons (each deamon corresponds to a single emulated node). We run an unmodified SLURM to manage the emulated nodes, but jobs are emulated using `sleep` commands. Thus, we had to modify the way SLURM collects energy measurements: as the platform is emulated, the energy consumed by each job is not measured through sensors, but instead we inject it directly based on the trace.

We use Light-ESP workload [27]. Light-ESP is based on ESP [28], a synthetic benchmark workload consisting of 230 jobs of 14 types. Light-ESP reduces the runtime of these jobs, so that the turnaround time of the whole log is 15 minutes. We repeated Light-ESP workload 4 times.

We compared the job stretches under four prioritization policies: FairShare (FS), EnergyFairShare (EFS), FS+EFS; and FIFO, in which jobs' priorities are based on their waiting time. As in the previous experiment, and as in a default SLURM configuration, an EASY backfilling algorithm is used to allocate processors to jobs.

The Light-ESP workload does not specify jobs' owners and jobs' energy consumption, thus we will study two cases. First, we add three users having the same workload, but varying energy efficiency. We expect that the energy-efficient user should have a smaller stretch than the inefficient one. Second, we add three users having different workload and the same energy efficiency. We expect that EFS will work in the same way as FS.

1) 3 users with the same workload and different energy efficiency: In this experiment, we add to the Light-ESP trace three users with the same workload but with different energy efficiency. User 1 is a user who optimized by 30% the energy efficiency of her job, user 2 is a normal user, and user 3 is inefficient (by 30%). Figure 2 presents results. In FS and FIFO policies, all users have roughly the same stretch which is expected as all users have the same workload. EFS and FS+EFS reward efficient users and punish the inefficient ones. User 3, who consumed the most energy, has also the highest

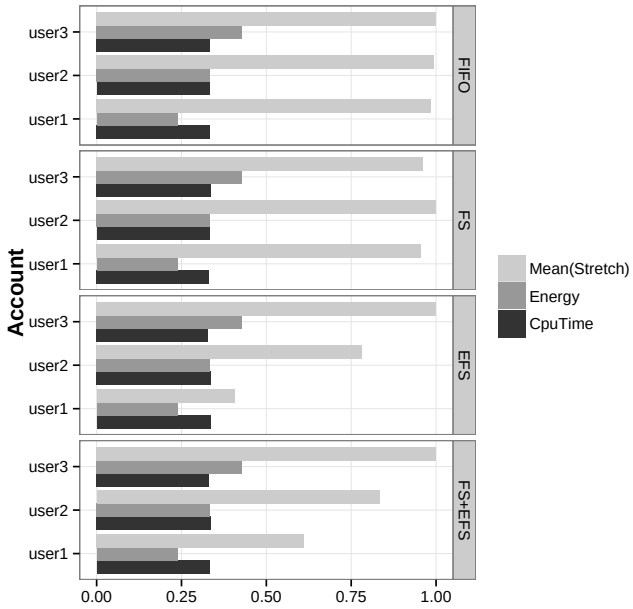


Fig. 2: SLURM implementation. Emulation of Curie cluster running 4 Light-ESP traces. Three users have the same workloads, but different energy efficiency. All values shown are normalized.

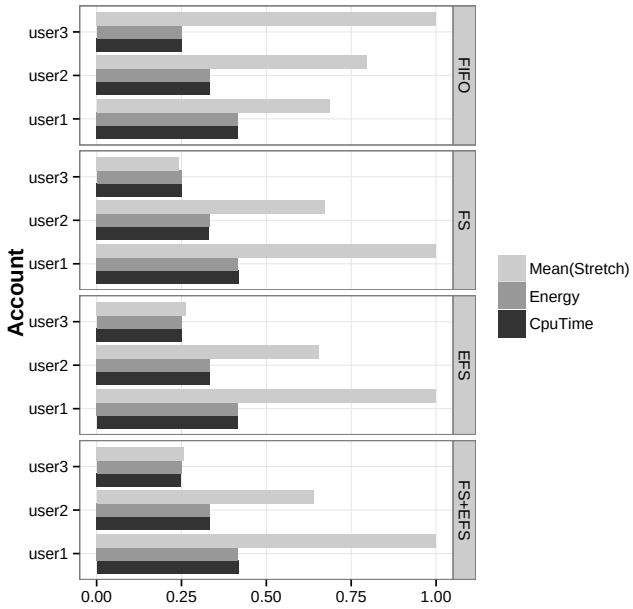


Fig. 3: SLURM implementation. Emulation of Curie cluster running 4 Light-ESP traces. Three users have different workloads, but the same energy efficiency. All values shown are normalized.

	Job Size (cores)	Number of Jobs	Run Time (sec)
Job Type	Job size for cluster of 180 cores (Fraction of job size relative to system size)/ Number of Jobs / Run Time (sec)		
A	12 (0.03125)	75	22s
B	12 (0.06250)	9	27s
C	96 (0.50000)	3	45s
D	48 (0.25000)	3	51s
E	96 (0.50000)	3	26s
F	24 (0.06250)	9	154s
G	36 (0.12500)	6	111s
H	36 (0.15820)	6	89s
I	12 (0.03125)	24	119s
J	24 (0.06250)	24	60s
K	24 (0.09570)	15	41s
L	36 (0.12500)	36	30s
M	48 (0.25000)	15	15s
Z	180 (1.0)	2	20s
Total Jobs / Theoretic Run Time	230 / 935s		

TABLE II: Synthetic workload characteristics of Light-ESP benchmark [27] as adapted for a 180 cores cluster with exclusive nodes allocations.

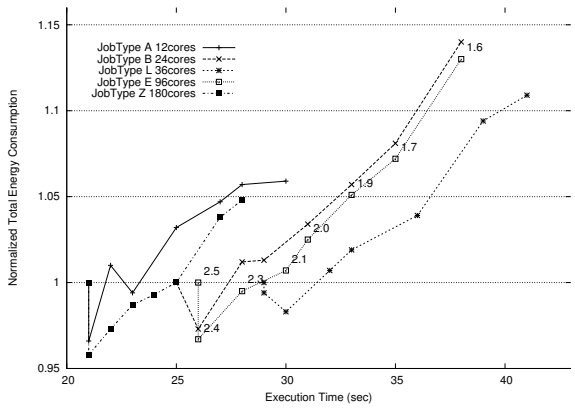


Fig. 4: Performance vs. energy tradeoffs for Linpack applications as calibrated for Light-ESP job types (table II) running on a 180-cores cluster at different frequencies.

mean stretch; whereas the efficient User 1 has a stretch reduced by roughly 60%. Compared to EFS, FS+EFS policy has smaller difference between each user's mean stretch because FS+ESF is a combination of the two above policies.

From a global point of view, policies perform equally well, as they all achieve a utilization of about 89%.

2) 3 users with different workload and the same energy efficiency: In this experiment, we add to the Light-ESP three users with different workload (and the same energy efficiency). Here, we want to show that our algorithm is equivalent to the classic FairShare, if jobs have the same energy efficiency. On Figure 3, we can see that, except for FIFO, each policy has the same results. User 3, the user with the highest workload (and also—the highest energy consumption) is penalized compared to other users. These variation of stretch for each user time does not change the global performance of the algorithm as each policy have the same mean utilization of 89%.

C. Experiments on a real platform

In this subsection we perform experiments on a real, small-scale cluster using monitored energy consumption data. Our objective is to evaluate the new EFS scheduling strategy as implemented upon SLURM and validate its effectiveness under real-life conditions. The experiments have been performed upon a small BULL cluster dedicated for R&D experiments. The cluster is composed of 16 nodes with Intel Xeon CPU E5649 (2 sockets/node and 6 cores/socket), 24GB of memory and Infiniband network. For the sake of the experiments, we installed our modified SLURM version, configured one dedicated SLURM controller, 15 compute nodes and activated the energy accounting framework with monitoring through IPMI method.

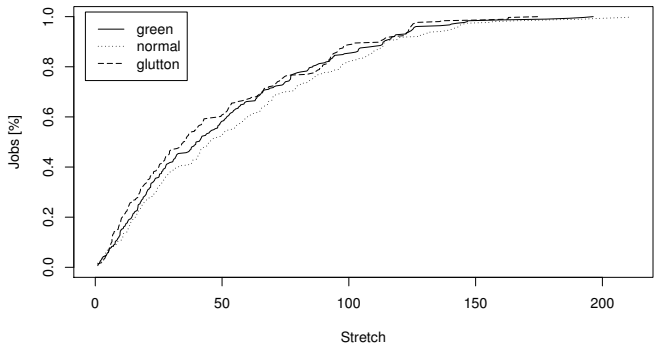
Our experiments follow the same guidelines as the emulation-based experiments in the previous subsection IV-B using the 4x Light-ESP benchmark as an input workload. Table II presents the characteristics of a single Light-ESP workload as adapted for the 180 cores of our cluster. There are two important differences compared to the workload used in IV-B: i) The energy consumption, within SLURM, is measured at the level of nodes (as described in Section III-D), which means that only exclusive allocations can be considered (i.e., two jobs cannot share a single node). Thus, the size of each job type in Table II is a multiple of 12 cores. ii) In order to observe actual energy consumption, instead of `sleep` commands, we have to execute real jobs. We used Linpack MPI applications that we calibrated to fit the target run-time of the benchmark.

Before starting the experiments we have also profiled Linpack for the different job classes in terms of energy-performance tradeoffs. Figure 4 provides graphs of the evolution of these tradeoffs when changing CPU frequencies of processors. We show just a representative part of table's II job types' observed behaviour. It is interesting to see that the lowest frequency is not the most energy-efficient. On the contrary: the most energy efficient CPU frequency varies between 2.3 Ghz and 2.2 Ghz for most cases; whereas the lowest CPU frequency (1.6 Ghz) is usually the most energy-consuming one. Note that this behavior might be specific to the type of the job used—a heavily-optimized, computationally-intensive Linpack. Also take into account that we are here effectively using a black-box approach, as we are scaling the whole application.

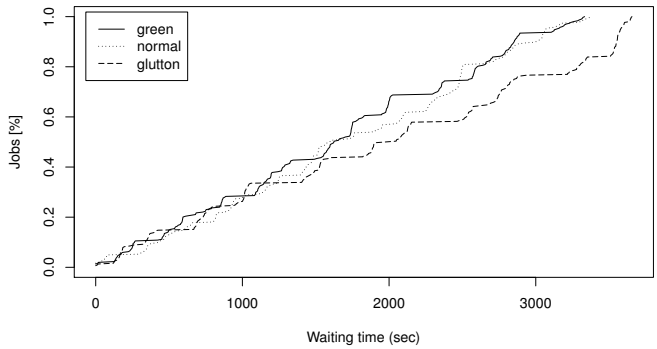
We divided the jobs from Light-ESP into three groups of equal sizes; then, we assumed that each group corresponds to a single user having different energy-efficiency.

The first user is "green"—she takes into account the tradeoffs analysis and adapts her CPU frequency to the optimal value per each job type. The second user is "normal"—he selects the highest frequency in order to optimize performance. Finally, the third user is "gluttonous"—he chooses the slowest CPU frequency resulting in the worst energy efficiency.

Each job is submitted in a particular moment in the workload. To be certain that the order of job submissions does not influences the results, we are launching 4 times the same workload (light-ESP); each time changing the attributed user

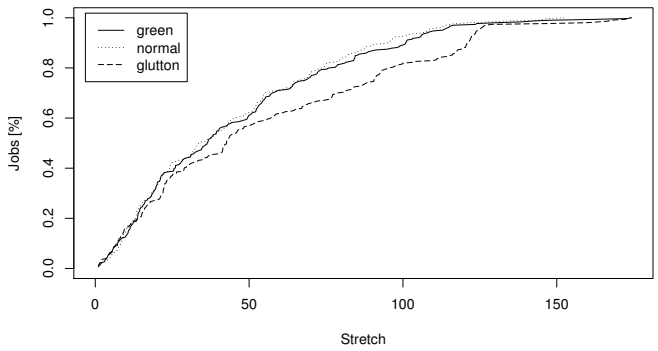


(a) CDF on Stretch with EFS

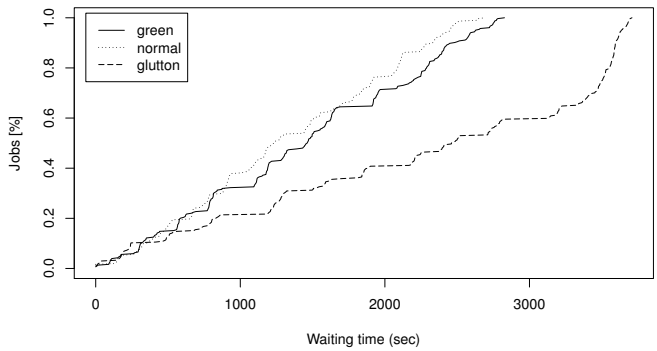


(b) CDF on Waiting time with EFS

Fig. 5: Cumulated Distribution Function for Stretch and Waiting time with SLURM EnergyFairShare policy running LightESP x4 workload with Linpack executions by 3 users with different energy efficiencies.



(a) CDF on Stretch with FS



(b) CDF on Waiting time with FS

Fig. 6: Cumulated Distribution Function for Stretch and Waiting time with SLURM FairShare policy running four LightESP workloads with Linpack executions by 3 users with different energy efficiencies.

per group of jobs.

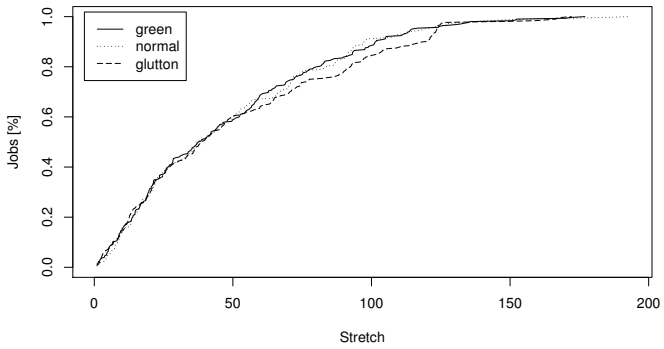
Figure 5a shows the cumulative distribution function (CDF) of stretch with only EFS activated. "Green" user's jobs have smaller stretch than normal user. However, it appears that the "green" and the "gluttonous" users have quite similar stretches. Even if this may seem surprising at first it is explained by the fact that in our context the gluttonous jobs are represented by the lowest CPU-Frequency. This results in high energy consumption because of very large run time. Hence even if gluttonous jobs have larger queue waiting times than green jobs because of EFS, they have also larger running times, which results in lower stretches.

Figure 5b shows the CDF of jobs' waiting time (the time a job spends in the queue) in the case of EFS policy. We can see that jobs are executed in groups according to the users' EFS factor.

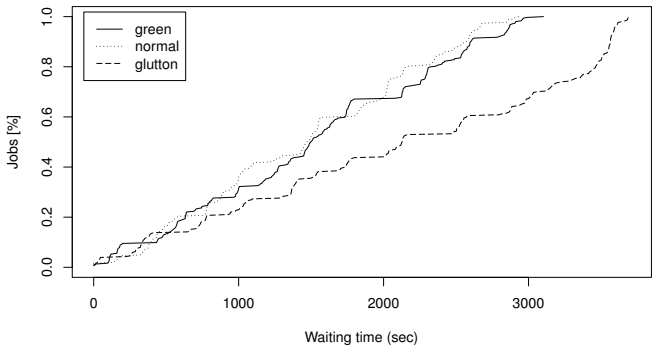
Jobs whom waiting time is up to approximately 1250 seconds have similar waiting time regardless of the energy efficiency. The influence of the efficiency is visible on longer-waiting jobs. When jobs are continously submitted, the EFS policy translates to a scheduling policy that executes a few jobs of one user; then a few jobs of another user, etc. When a user becomes the one with the minimal EFS factor, the remaining user's jobs are started as long as the user's EFS factor remains lower than the other two users. Eventually, the energy consumed by the newly started jobs adds up, and another user becomes the one having the minimal factor. The number of executed jobs per such "turn" depends on the consumed energy. That is why the gluttonous user eventually is waiting much longer than the other two users. The green users' jobs have slightly smaller waiting time than the normal users' jobs.

Figure 6 shows the CDF of stretch and waiting times with the same workload and jobs execution as previously, but using the original FairShare (FS). In this case we consider only CPU-time for prioritization. Both in terms of stretch and waiting times we can see that "normal" user's jobs have the best results since their jobs have the optimal performance. "Green" user's jobs have quite good results whereas "gluttonous" user's jobs have the worst result because of large run times.

In the following experiment we have activated both FS along with EFS with equal weights within SLURM. This

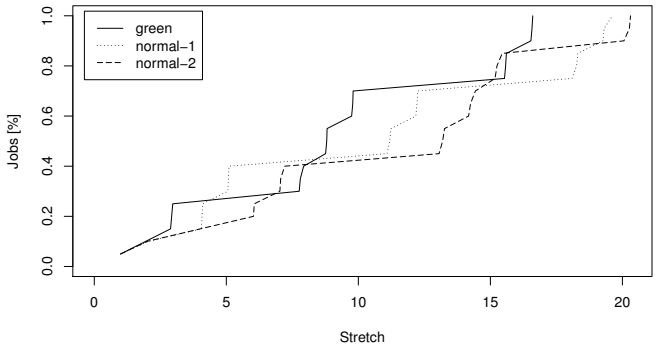


(a) CDF on Stretch with FS + EFS

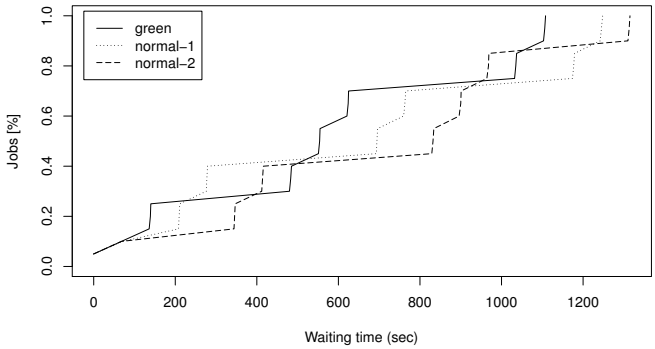


(b) CDF on Waititng time with FS + EFS

Fig. 7: Cumulated Distribution Function for Stretch and Waiting time with SLURM FairShare plus EnergyFairShare policies running four LightESP workloads with Linpack executions by 3 users with different energy efficiencies.



(a) CDF on Stretch with EFS



(b) CDF on Waiting time with EFS

Fig. 8: Cumulated Distribution Function for Stretch and Waiting time with SLURM EnergyFairShare policy running a submission burst of 60 similar jobs with Linpack executions by 1 energy-efficient and 2 normal users

means that both CPU-time and energy consumption can play equal role in the calculation of job's priority. Figure 7 shows the CDF of stretch and waiting times with light-ESP x4 workload with FS+EFS policies. "Gluttonous" user's jobs have an important disadvantage when compared to the other two groups, especially in terms of waiting times. In contrast with 5a, figure 7a shows a significant degradation for "gluttonous" user's jobs and this is because the waiting times are much longer in the FS+EFS case as we can see in 7b. Furthermore, we can observe that "green" and "normal" user's jobs provide similarly good results. When "green" jobs gain in energy, "normal" jobs gain equally in CPU-time. However, we can observe a slight advantage of "normal" user's jobs. This can be explained by the fact that the difference in gained CPU-time of "normal" jobs are more noticeable than the gains in energy consumption by the "green" jobs.

Finally we performed another experiment with a simple workload of 60 jobs executing the same Linpack application upon 60 cores each. The jobs were separated into 3 groups where one group is launched as energy efficient with the optimal CPU-Frequency (2.3GHz) and the other two have been launched both with normal characteristics (one with larger CPU-Frequency 2.5GHz and one using on-demand governor). All the jobs have been launched simultaneously on the cluster. The difference with previous figures (in addition to having of 2 normal users), was that we have calibrated Linpack to be executed with good energy-efficiency when running with DVFS of 2.3 GHz. The results are presented in form of CDF on stretch and waiting times in figures 8a and 8b. The figures show a significant improvement in both stretch and waiting times for the green user's jobs. Hence, in these figures, the higher energy-efficiency of the green jobs is rewarded with optimized stretch and waiting times.

V. Conclusion and future works

In this paper, we proposed EnergyFairShare, an algorithm to prioritize jobs based on their owners' past energy usage. The main goal of our policy is to explicitly manage what is currently an important cost of running an HPC resource: the electricity. In general, the more electricity a user consumed in the past, the lower is the priority of her future jobs. We implemented the algorithm in a simulator, and as a plugin for

SLURM, a popular HPC resource manager. Our implementation of scheduling policies will appear in the upcoming stable release of SLURM (version 15.08). We verified experimentally that more energy-efficient jobs have lower stretches. We claim that EnergyFairShare should motivate users to make their jobs more energy-efficient, in the same way as FairShare incentives users to make their jobs more CPU-runtime efficient—however, to test this claim, our mechanism would need to be used in a production system.

Utilization of any resource implies energy consumption, thus a possible objection to our work might be that we penalize "large" jobs that should be executed anyway (after all, the interest in doing HPC are large-scale calculations). However, we use the same principle as the one already used for managing CPU-seconds—FairShare. Scheduling policies employing FairShare are very common in existing HPC centers; indeed, FairShare policies penalize users submitting large jobs; but if a particular user demonstrates that the size of her job is a consequence of a true, scientific need (and not — inefficiency), the administrators can increase her target share. Same argument applies to EnergyFairShare.

Not all resources consume the same amount of energy, thus some resources may be more "costly" to users in terms of EFS. We claim that this heterogeneity should push users to run jobs on resources that are efficient for these jobs.

Of course, a scheduler-level prioritization mechanism is not sufficient to make users save energy. First, users need to be aware of the total energy consumption of each of her jobs; and, perhaps additionally, about their average energy efficiency (number of Watts consumed during an average CPU-second), compared to other jobs in the system. EFS already stores the information needed to compute these values; they can be presented in addition to existing accounting. Second, energy profilers should help to tune jobs for energy-efficiency (just as standard profilers help to tune for the run-time).

Finally, to be energy-efficient, we must first precisely measure the consumed energy. Currently, it is possible to measure either the whole node (which is a problem when a node is shared by a few jobs), or some components (CPU, GPU)—but not the others (network, memory, storage). However, as energy efficiency is a key to performance and to low running costs, we envision that more and more precise measures will be available.

Acknowledgements: We thank the providers of the workload logs we used in simulation: Joseph Emeras (Curie), Ciaron Linstead (PIK IPLEX) and Victor Hazlewood (SDSC SP2). The work is partially supported by the ANR project called MOEBUS and by the Polish National Science Center grant Sonata (UMO-2012/07/D/ST6/02440).

REFERENCES

[1] B. Subramaniam and W. Feng., "The green index: A metric for evaluating system-wide energy efficiency in HPC systems," in *IPDPS Workshops*, 2012.

[2] J. Dongarra et al., "The international exascale software project roadmap," in *International Journal of High Performance Computing Applications*, 2011.

[3] Y. Georgiou, T. Cadeau, D. Glesser, D. Auble, M. Jette, and M. Hautreux, "Energy accounting and control with SLURM resource and job management system," in *ICDCN*, 2014.

[4] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM Journal on Scientific Computing*, 2012.

[5] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple linux utility for resource management," in *JSSPP, Proc.*, 2003.

[6] Intel, "Intelligent platform management interface specification v2.0."

[7] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann, "Power-management architecture of the Intel microarchitecture code-named Sandy Bridge," *IEEE Micro*, 2012.

[8] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[9] R. Basmadjian and H. de Meer, "Evaluating and modeling power consumption of multi-core processors," in *Future Energy Systems: Where Energy, Computing and Communication Meet (e-Energy), 2012 Third International Conference on*. IEEE, 2012.

[10] G. L. Tsafack Chetsa, G. Da Costa, L. Lefevre, J.-M. Pierson, O. Ariel, and B. Robert, "Energy aware approach for HPC systems," in *High-Performance Computing on Complex Environments*, 2014.

[11] M. Assunção, J.-P. Gelas, L. Lefèvre, and A.-C. Orgerie, "The green Grid'5000: Instrumenting and using a grid with energy sensors," in *Remote Instrumentation for eScience and Related Aspects*, 2012.

[12] F. Rossigneux, J.-P. Gelas, L. Lefevre, and M. D. de Assuncao, "A generic and extensible framework for monitoring energy consumption of OpenStack clouds," *arXiv preprint arXiv:1408.6328*, 2014.

[13] Daniel Hackenberg et al., "Hdeem: High definition energy efficiency monitoring," in *2nd International Workshop on Energy Efficient Supercomputing*, 2014.

[14] Giorgio Luigi Valentini et al., "An overview of energy efficiency techniques in cluster computing systems," *Cluster Computing*, 2013.

[15] S. Mittal, "Power management techniques for data centers: A survey," *arXiv preprint arXiv:1404.6681*, 2014.

[16] S. Albers, "Energy-efficient algorithms," *Communications of the ACM*, 2010.

[17] N. Kappiah, V. W. Freeh, and D. K. Lowenthal, "Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005.

[18] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *NSDI*, 2011.

[19] S. Kleban and S. Clearwater, "Fair share on high performance computing systems: what does fair really mean?" in *CCGrid, Proc.*, 2003.

[20] D. Jackson, Q. Snell, and M. Clement, "Core algorithms of the Maui scheduler," in *JSSPP, Proc.*, 2001.

[21] D. Klusacek and H. Rudova, "Performance and fairness for users in parallel job scheduling," in *JSSPP, Proc.*, 2013.

[22] J. Emeras, V. Pinheiro, K. Rzadca, and D. Trystram, "Ostrich: Fair scheduling for multiple submissions," in *PPAM, Proc.*, 2014.

[23] D. Klusaček and H. Rudova, "Multi-resource aware fairsharing for heterogeneous systems," in *JSSPP, Proc.*, 2014.

[24] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang, "Multiresource allocation: Fairness-efficiency tradeoffs in a unifying framework," *IEEE/ACM Trans. Netw.*, 2013.

[25] A. Mu'alem and D. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *Parallel and Distributed Systems, IEEE Transactions on*, 2001.

[26] Pyss, "https://code.google.com/p/pyss/."

[27] Y. Georgiou and M. Hautreux, "Evaluating scalability and efficiency of the resource and job management system on large HPC clusters," in *JSSPP, Proc.*, 2013.

[28] A. T. Wong, L. Oliker, W. T. Kramer, T. L. Kaltz, and D. H. Bailey, "ESP: A system utilization benchmark," in *SuperComputing*, 2000.