PShifter: Feedback-based Dynamic Power Shifting within HPC Jobs for Performance

Neha Gholkar[1], Frank Mueller[1], Barry Rountree[2], Aniruddha Marathe[2]
[1]North Carolina State University, USA, ngholka@ncsu.edu, mueller@cs.ncsu.edu
[2]Lawrence Livermore National Laboratory, USA, rountree1@llnl.gov, marathe1@llnl.gov

ABSTRACT

The US Department of Energy (DOE) has set a power target of 20-30MW on the first exascale machines. To achieve one exaFLOPS under this power constraint, it is necessary to manage power intelligently while maximizing performance. Most production-level parallel applications suffer from computational load imbalance across distributed processes due to non-uniform work decomposition. Other factors like manufacturing variation and thermal variation in the machine room may amplify this imbalance. As a result of this imbalance, some processes of a job reach the blocking calls, collectives or barriers earlier and wait for others to reach the same point. This waiting results in a wastage of energy and CPU cycles which degrades application efficiency and performance.

We address this problem for power-limited jobs via Power Shifter (PShifter), a dual-level, feedback-based mechanism that intelligently and automatically detects such imbalance and reduces it by dynamically re-distributing a job's power budget across processors to improve the overall performance of the job compared to a naïve uniform power distribution across nodes. In contrast to prior work, PShifter ensures that a given power budget is not violated. At the bottom level of PShifter, local agents monitor and control the performance of processors by actuating different power levels. They reduce power from the processors that incur substantial wait times. At the top level, the cluster agent that has the global view of the system, monitors the job's power consumption and provides feedback on the unused power, which is then distributed across the processors of the same job. Our evaluation on an Intel cluster shows that PShifter achieves performance improvement of up to 21% and energy savings of up to 23% compared to uniform power allocation, outperforms static approaches by up to 40% and 22% for codes with and without phase changes, respectively, and outperforms dynamic schemes by up to 19%. To the best of our knowledge, PShifter is the first approach to transparently and automatically apply power capping non-uniformly across processors of a job in a dynamic manner adapting to phase changes.

ACM Reference Format:
Neha Gholkar[1], Frank Mueller[1], Barry Rountree[2], Aniruddha Marathe[2] [1]North Carolina State University, USA, ngholka@ncsu.edu, mueller@cs.ncsu.edu [2]Lawrence Livermore National Laboratory, USA, rountree1@llnl.gov, marathe1@llnl.gov . 2018. PShifter: Feedback-based Dynamic Power Shifting within HPC Jobs for Performance. In *HPDC '18: The 27th International Symposium on High-Performance Parallel and Distributed Computing, June 11–15, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3208040.3208047

1 INTRODUCTION

A hardware-overprovisioned system [28, 36–38] consists of more hardware or nodes than can be powered at thermal design power (TDP) simultaneously. Depending on the job's characteristics (e.g., memory usage, communication) some jobs achieve high performance on fewer nodes at high power while others achieve high performance on larger numbers of nodes at low power. Hence, on an overprovisioned system, an intelligent power scheduler that schedules power across jobs, is a must in addition to a conventional scheduler. A power scheduler assigns power budgets to jobs such that the system's power consumption at any point in time does not violate the budget, e.g., the DOE limit. Each job needs to achieve the best possible performance while adhering to its power budget.

A naïve approach of enforcing a job's power constraint is to distribute power *uniformly* across all the nodes of the machine. We call this uniform power (UP). UP can be enforced by statically constraining the power consumption of nodes to $\frac{Job's PowerBudget}{N}$, where N is the number of nodes of a job. While UP enforces a job-level power bound, it may lead to sub-optimal performance as it does not handle static or dynamic load imbalance induced due to non-uniform workload distribution across nodes [15]. Processor variation [32] may worsen the performance further. Prior work, PTune [15] and Conductor [26], are other power management solutions that also aim at maximizing the performance of a job under a strict power budget. PTune is a processor variation-aware static job power management solution but it does not handle the load imbalance. Conductor, is a dynamic job power management solution. While Conductor shifts power on-line by detecting the critical path to speed up the job, our evaluation shows that it has high overheads for load-imbalanced codes that offset its performance gains.

We address the inadequacies of existing power assignment solutions with Power Shifter (PShifter), a runtime system that maximizes job's performance without exceeding its assigned power budget. Determining a job's power budget is beyond the scope of this paper. PShifter is a hierarchical closed-loop feedback controller that makes measurement-based power decisions at runtime and adaptively. At the top level, the cluster agent monitors the power consumption of the entire job. It opportunistically improves the job's performance by feeding its unused power budget back into the job. Each socket is periodically monitored and tuned by a local agent. A local agent is a proportional-integral (PI) feedback controller that strives to reduce the energy wastage by its socket. It gives up socket power when its socket is an "early bird" that waits at blocking calls and collectives. The cluster agent senses this power dissipation within a job in its monitoring cycle and effectively redirects the dissipated power to where it can best improve the overall performance of the job (i.e., reduce the critical path). In experiments, PShifter achieves a performance improvement of up to 21% and energy savings of up to 23% compared to the naïve approach.

Unlike prior work that was agnostic of phase changes in computation, PShifter is first to transparently and automatically apply power capping non-uniformly across nodes of a job in a dynamic manner adapting to phase changes. It could readily be deployed on any HPC system with power capping capability without any modifications to the application's source code.

Contributions:

- We propose the design and implementation of PShifter.
- The major contribution of PShifter is smoothing computational imbalance within a job by dynamically moving the required amount of power from low-computation to high-computation sockets.
- We evaluate PShifter using two of the Mantevo applications [35], miniFE and CoMD, and a production application, ParaDis [5] and compare results to prior work and a status quo scheme.
- We present the comparison of PShifter with prior work [15, 26] that is closest to PShifter.

2 OVERVIEW

Today's systems are worst-case power-provisioned, which means that enough electrical capacity is purchased to allow all the nodes to operate at full power simulataneously. However, only a few "heroic" codes ever use anywhere close to TDP, and most mission-critical codes consume around 60% [29]. Leveraging this knowledge, researchers have proposed hardware-overprovising [28, 36–38] which promotes procurement of larger systems with limited power. A hardware-overprovisioned system consists of more hardware or nodes than can be powered at TDP simultaneously. On such a system, the job scheduler determines the power budget and the number of nodes for each of the jobs based on its characteristics such that all the executing jobs can complete without exceeding the provisioned power capacity.

The Uniform Power scheme (described in Sec. 1) is a naïve approach of enforcing a job's power budget. From Sandy Bridge processors onward, Intel provides the Running Average Power Limiting (RAPL) [21] interface that allows the programmer to measure and bound the power consumption of the package (PKG) and the memory (DRAM). Here, package, also called socket, is a single multi-core Intel processor. To make our work feasible, we limited ourselves to power consumed by the sockets.

UP has two major drawbacks. First, processors on a cluster exhibit variations in power efficiency translating to variation in performance [15, 32] that make the cluster non-homogeneous. Second, in practice, parallel applications tend to experience computational load imbalance across their processes (worker threads) due to uneven work distributions. Due to this imbalance, some processes finish computation early and wait for others at barriers, collectives or blocking calls even on homogeneous systems. A non-homogeneous system under a uniform power scheme can actually *worsen* performance depending on if an overloaded process is mapped to an inefficient socket, which lengthens its computation phases. This leads to longer barrier wait times for underloaded processes. Such waiting results in unused CPU cycles at barriers and other collectives, which further contributes to energy wastage.

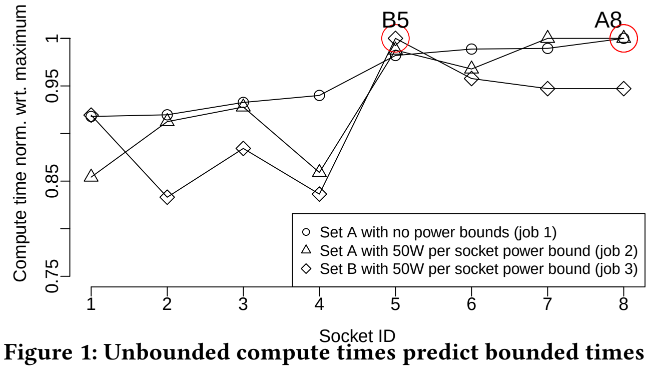

Figure 1: Unbounded compute times predict bounded times poorly.

The application's computation time profile also differs across disjoint sets of sockets under a power bound making it difficult to derive a static solution. This is depicted in Fig. 1. The x-axis represents socket identifiers and the y-axis represent computation time between two synchronization calls normalized with respect to the maximum compute time across all sockets of a job (lower is faster). The application runs on 8 sockets. The socket finishing last, i.e., the socket on the critical path, is circled.

The graph shows three jobs running on two distinct sets (A and B) of sockets. The job on Set A with no power bounds is represented by circles and A8 is the performance bottleneck. The job with an average socket power bound of 50W on set A is represented by triangles and A8 is its performance bottleneck. The job on Set B with an average socket power bound of 50W is represented by diamonds and B5 is its performance bottleneck. The mapping of processes to socket IDs is indentical in all three jobs.

We observe the following:

- Uneven work distribution in a parallel job leads to different compute times across sockets leading to wait times and wasted cycles and energy.
- The compute time curve with no power bounds (uniform performance) is different from that with power bounds on sockets. For example, under power bounds, socket 4 finishes before sockets 2 and 3 unlike the no power bounds case.
- The compute time curve with identical power bounds on two different sets of sockets, A and B, is different. The performance bottleneck shifts from socket A8 to socket B5. This is because B5 (set B) is more power efficient than socket A8 (set A). Hence, under identical power bounds, the former completes the same amount of computation significantly faster than the latter.

We propose PShifter, a runtime system that addresses these drawbacks and maximizes job's performance without exceeding its assigned power budget.

3 DESIGN

PShifter is a hierarchical, closed-loop feedback controller that operates at runtime alongside the job. The key idea is to make measurement-driven decisions dynamically about moving power within a job from where it is wasted to where it is required (on the performance-critical path).

3.1 Closed-loop feedback controller

The general idea of a closed-loop feedback controller is depicted in Fig. 2. *System* is the component whose characteristics are to be monitored and actuated by the controller. The current state of the system is defined in terms of the *process variable* (PV). The desired state of the system, PV=K, where K is a constant, is called the setpoint (SP). It is input to the feedback controller. A feedback-control consists of three main modules, viz., sensor, control signal calculator, and actuator.

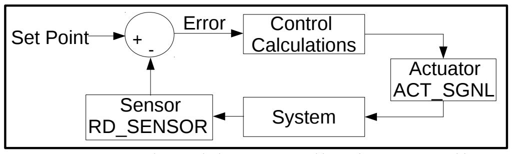

Figure 2: Closed-loop Feedback Controller

Sensor: The sensor periodically monitors the state of the system defined in terms of process variable. This is indicated by RD_SENSOR.
Control Signal Calculator: The error (err) in the system state is calculated as the difference between the setpoint and the measured process variable. The feedback controller calculates the feedback to the actuator as a function of this error. It also takes into account the history of error values. The output of the feedback-control loop is calculated as the sum of the previous values of the actuator and the calculated feedback.
Actuator: The actuator applies the calculated signal to the system. This is indicated by ACT_SGNL.

3.2 PShifter

PShitfter consists of a dual-level hierarchy of closed-loop feedback controllers, a local agent and a cluster agent. Fig. 3 depicts the high-level architecture of the controller and the mapping of its components to the components of the underlying system.

Fig. 3(a) shows the architecture of a typical HPC cluster. It consists of multiple server nodes. Each server node hosts one or more sockets. Our server nodes have two sockets, each hosting a single 12-core processor. Multiple parallel jobs spanning across one or more nodes can run on a cluster, e.g., job1 runs on the top 4 nodes while job2 runs on bottom 4 nodes. An instance of PShifter runs alongside each job and enforces its power budget while improving job performance under the power constraint.

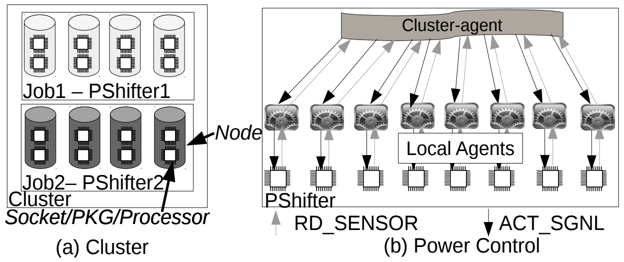

Figure 3: PShifter Overview

PShifter's controller hierarchy is depicted in Fig. 3(b). It consists of a cluster agent at the root and several local agents at the leaves. At the bottom level, several local agents monitor the performance and manage the energy consumption of the individual sockets of the job. At the top level, the cluster agent overlooks the power consumption of the entire job.

Initialization. At job initialization, the cluster agent enforces a job's power budget by uniformly distributing the power across its sockets. This is the system state established by the naïve scheme. It enforces the job level power budget but it does not address the problem of performance degradation due to imbalanced jobs. Starting form this initial state, PShifter (local and cluster agents) gradually moves power within the job to change the unbalanced performance state of the system to a desired, more balanced performance.

Local agent. Local agents are the leaf nodes in PShifter's hierarchy and implement a closed-loop feedback controller. A local agent monitors and controls performance and energy consumption of a unique socket in the job allocation. There is one local agent per socket in the job allocation. The local agents are invoked periodically and asynchronously. The process variable, PV, for the local agent is defined as the ratio of the computation time to the total time (including computation, wait, and communication time) between two subsequent invocations.

$PV = \frac{ComputationTime}{TotalTime}$

It is a measure of the socket's computational load. In other words, it measures the proportion of the total time spent doing useful work. Sockets with comparatively lower PV are called *underloaded sockets* while sockets with higher PV re called *overloaded sockets*. Notice that computation time includes memory access time. Thus, the process variable captures the memory-boundedness of the job.

A local agent uses the power cap of the associated socket as its actuator. Each socket hosts multiple (12) processes, each pinned to a unique core. Pinning processes to cores avoids unnecessary overhead of Linux process migrations and longer memory access delays due to remote NUMA accesses. The PV_p is measured by each process and for each socket's local agent. The process with the maximum PV_p is called the representative process of the socket and the socket's PV is equal to the representative process's PV_p. The processes that have shorter computation tasks finish early and wait for other processes (with longer computation tasks) before engaging in communication or synchronization. These wait times lead to wastage of power that is not utilized for any work (computation). The PV of a socket with "early bird" processes tends to be much less than that of sockets with a bottleneck process, which tend to have a PV of almost one (as communication takes some finite time). The goal of the local agent is to maximize the PV (Note: $PV \leq 1$) and thus to minimize the wait time. The setpoint (SP) is initialized to 0.95. It is experimentally determined. The local agent strives to achieve this setpoint by lowering the power of the socket with PV < 0.95. This slows down the processes that originally had shorter computation tasks and subsequently reduces their wait times in future iterations. It is important to note that local agents make decisions based only on local information, i.e., they do not need to communicate or synchronize with others.

The local agent is implemented as a proportional-integral (PI) controller. The PI controller makes power decisions dynamically at runtime based on measured values of the process variable. The PI feedback from the local agent is calculated as

$fb_{local\_agent} = Pterm + Iterm,$

where Pterm denotes the proportional term and Iterm stands for integral term. Pterm is calculated as

$Pterm = Kp * e(t),$

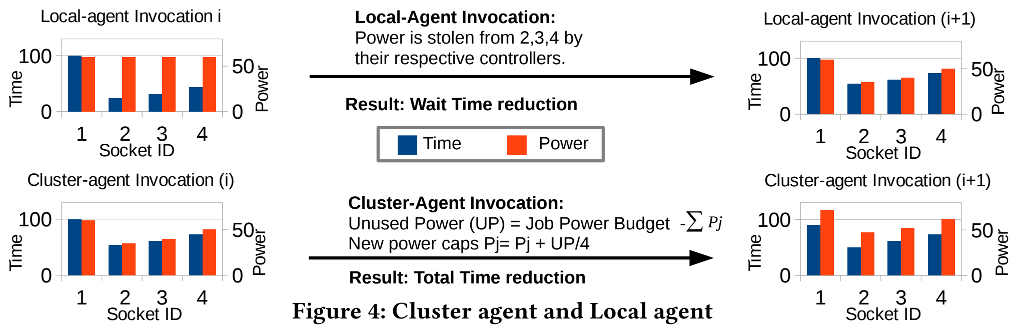
Local-agent Invocation i
Time
Power
Socket ID
Local-Agent Invocation:
Power is stolen from 2,3,4 by their respective controllers.
Result: Wait Time reduction
Time
Power
Local-agent Invocation (i+1)
Cluster-agent Invocation (i)
Cluster-Agent Invocation:
Unused Power (UP) = Job Power Budget -∑ Pj
New power caps Pj= Pj + UP/4
Result: Total Time reduction
Cluster-agent Invocation (i+1)

Figure 4: Cluster agent and Local agent

where Kp is a constant. Iterm is calculated as
$Iterm = Ki * \int e(t)dt$,
where Ki is a constant. The error, (e(t)) at time t, is calculated as
$err = PV - SP$.

The constants were application specific (but vary only 5% between applications) and were determined experimentally, a common method in feedback-control systems: We set the integral gain, Ki, to zero and increase the proportional gain, Kp, until the outputs oscillate. We then increase Ki to reduce the steady state error to an acceptable level of 5%.

Pterm accounts for the current error and calculates a response proportional to this error. Iterm is calculated as the product of aggregated past errors over time and the constant Ki. Iterm is dependent on the magnitude of past errors and the time for which they stay uncorrected. If the output generated by the proportional term is small and does not reach the setpoint over multiple invocations, the error aggregated over several invocations helps in strengthening the output (larger Iterm) and, in turn, approaches the setpoint faster. This may cause the controller output to overshoot the setpoint. Hence, we use a bandpass filter to limit Iterm. Error values are accumulated for Iterm only until it reaches the upper or lower (error can be positive or negative) saturation point of the filter. The power cap of a socket is the actuator. The output of the local agent or the new power cap is calculated as
$Pcap_{(current)} = Pcap_{prev} + fb_{local\_agent}$,
where $Pcap_{prev}$ is the previous power cap of the socket and fb_{local_agent} is the calculated feedback.

The impact of power modulation on the socket's performance or PV depends on the memory-boundedness and the CPU-boundedness of a job. As the local agent monitors PV and keeps track of the error history as it sets power caps at each invocation, it indirectly learns about the nature of the job. It incorporates this knowledge into feedback in the form of Iterm.

Cluster agent. The PV of the cluster agent is the job's power consumption measured using Intel's RAPL interface, i.e. the aggregate power consumption of all of its sockets. The cluster agent's SP is set to the job's power budget (P_{job_budget}).
$err = SP - PV = P_{job\_budget} - \sum_{i=1}^{i=N} P_i$,
where P_i is the measured power consumption of the i^{th} socket. Note that the error is indicative of a job's unused power (P_{job_unused}) budget. The cluster agent distributes this unused power uniformly across all sockets. The feedback from the cluster agent ($fb_{cluster\_agent}$) is calculated simply as
$fb_{cluster\_agent} = \frac{P_{job\_unused}}{N}$,
where N is the total number of sockets of the job.

The cluster agent and the local agents share their actuators, i.e., the cluster agent also uses the power actuators of the sockets. When the cluster agent is invoked, it overwrites the power caps of sockets. The new power caps are calculated as $Pcap_{current} = Pcap_{prev} + fb_{cluster\_agent}$. The cluster agent is invoked less frequently than the local agent giving the local agent multiple opportunities to give up just enough power to reach the local setpoint. When the cluster agent is invoked, it effectively feeds the unused power back into the job by redistributing it uniformly across sockets. This step results in redirecting some power from the sockets that have shorter computation to the sockets that have longer computation (bottleneck processes that determine the completion time of the job) in every invocation. This opportunistically speeds up bottleneck processes and reduces the overall completion time of the job.

Fig. 4 summarizes the design of PShifter. The local agents of sockets 2, 3, and 4 reduce the power of the sockets to slow down the processing resulting in reduced wait times. When the cluster agent is invoked, it calculates the unused power of the job and distributes this uniformly across all of its sockets. As a result, socket 1, which never gave up any power, now gets some additional power to finish its computation faster, resulting in an overall improvement in performance.

4 IMPLEMENTATION AND EXPERIMENTAL FRAMEWORK

Our solution was developed on the *Catalyst* cluster at Lawrence Livermore National Laboratory (LLNL). It is a 324-node Intel Ivy Bridge cluster. Each node has two 12-core Intel(R) Xeon(R) E5-2695 v2 @ 2.40GHz processors and 128 GB of memory. Each MPI job runs on a dedicated set of nodes on this cluster. No two jobs share nodes. For the power measurement and actuation, we leverage Intel's Running Average Power Limiting (RAPL) feature [21]. From Sandy Bridge processors onward, Intel supports this interface that allows the programmer to measure and constrain the power consumption of the package (PKG) by writing a power limit into the RAPL model specific register (MSR). Here, a package is a single multi-core processor chip or a socket. RAPL is implemented in hardware. It guarantees that the power consumption stays at the power limit specified by the user. The msr-safe kernel module installed on this cluster enabled us to read from and to write to the Intel RAPL model specific registers in userspace via the libmsr library [39].

We used MVAPICH2 version 1.7. The codes were compiled with the Intel compiler version 12.1. PShifter is a library that can be linked with the application. It is implemented using the MPI standard profiling interface[27] (PMPI). The feedback controllers are

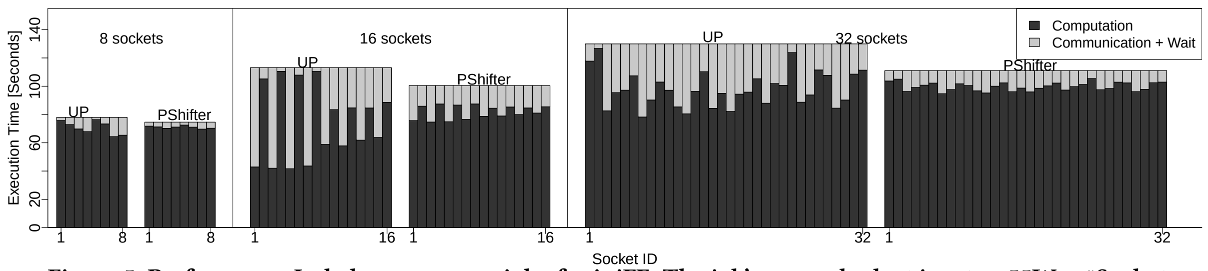

Figure 5: Performance Imbalance across a job of miniFE. The job's power budget is set as 55W × #Sockets

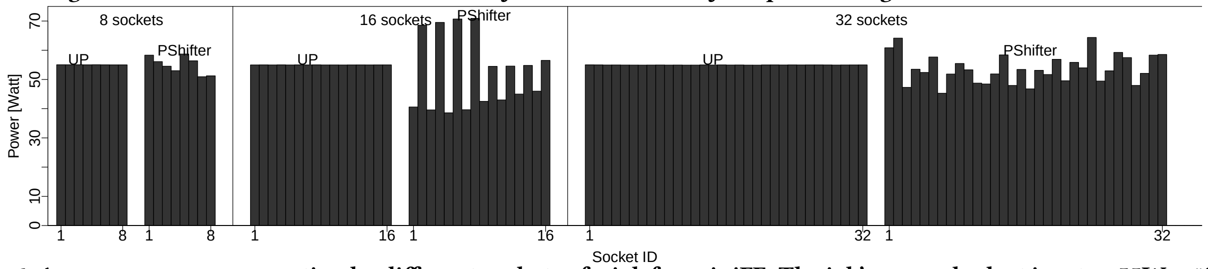

Figure 6: Average power consumption by different sockets of a job for miniFE. The job's power budget is set as 55W × #Sockets

called in the wrapper functions of MPI calls, which are invoked by an MPI application. Hence, our solution does not require any modifications to the application. Applications only need to be linked to our library. We pin each MPI process to different cores of the job's sockets. The term process is used to refer to an MPI process.

At initialization, i.e., in the MPI_Init wrapper, PShifter sets the RAPL power cap of all the sockets within a job to $\frac{Job'sPowerBudget}{N}$. It creates two types of MPI sub-communicators, viz., MPI_LOCAL_COMM per socket and MPI_CLUSTER_COMM per job using MPI_Comm_split. Each MPI_LOCAL_COMM communicator consists of all the processes pinned to the cores on the same socket. The process pinned to core 0 acts as sub-root (sub-rank 0) and runs the local agent. The MPI_CLUSTER_COMM consists of all the local agents with the local agent at MPI rank 0 acting as the root or the cluster agent. Each process records its computation time and the total time (computation+wait time). It then calculates its PV at every invocation. The local agent of every socket selects the process with maximum PV across all its processes as a representative. It also measures the power consumption of its socket. The local agent computes fb_{loc} using this PV and sets the new power cap for its processor. To set the power cap, it calls set_rapl_limit of the libmsr library.

The cluster agent invocation involves three steps. First, it measures the power consumption of the job by aggregating the power consumption across its sockets. This is done using MPI_Reduce over the MPI_CLUSTER_COMM communicator. The feedback is calculated and broadcasted (using MPI_Bcast) by the cluster agent to all the local agents that enforce the new power caps.

5 EXPERIMENTAL EVALUATION

We evaluated PShifter with MiniFE and CoMD proxy applications from the Mantevo [17] benchmark suite, and a production application, ParaDiS [5]. miniFE is representative for unstructured finite element codes. CoMD is a proxy application for molecular dynamics codes. Explicit load imbalancing was turned on for these proxy applications. ParaDiS [5] is a dislocation dynamics simulation code. We weakly scaled the inputs to miniFE and CoMD by increasing the input sizes proportionally to the number of nodes. For ParaDiS we used two LLNL inputs, small scale ($\leq$ 384 cores) and large scale ($>$ 384 cores). We weakly scaled each of these two inputs as we ran them on up to 32 and 256 sockets, respectively.

We used the MPI versions of these codes in our experiments. We report performance in terms of job completion time in seconds, average power in Watts, and energy in Joules. The reported numbers are averages across five runs. with a maximum standard deviation of 7%. The baseline for evaluation is uniform power (UP), where the job's power budget is distributed uniformly across all the sockets of a job. The maximum socket power consumption across our applications was observed to be 90, i.e., none of these benchmarks, would ever exceed 90W on a socket. Hence, we show evaluation results for power constraints ranging from 90% to 60% (80W to 55W, respectively) of maximum power.

5.1 Comparison with Uniform Power (UP)

Fig. 5 shows the performance imbalance that persists within 8, 16, and 32 socket jobs of miniFE. The job's power budget is set to 440W, 880W, and 1760W for 8, 16, and 32 socket jobs, respectively. The y-axis represents execution time of the job in seconds. Each stacked bar in the plot represents the computation (gray) and non-computation (white) time for each socket's representative process. The total height of the bars (gray+white) indicates the completion time of the job. For each job size, there are two groups of bars, one corresponding to the UP scheme and the other corresponding to the PShifter scheme as labeled in the plots. The x-axis represents the socket IDs within each job. It can be observed that some sockets spend more time in the computation phase than other sockets causing performance imbalance within the job. This imbalance can be quantified as $I = \frac{max(C_i)-min(C_i)}{mean(C_i)}$, where C_i is the computation time of the i^{th} socket's representative process.

Observation 1: PShifter moves power from underloaded sockets to overloaded sockets.

For example, in case of the 16 socket job, it can be observed that under UP, even numbered sockets have longer computation phases and, hence, high PVs (overloaded sockets) than odd numbered sockets (underloaded sockets). PShifter effectively moves power from underloaded sockets to overloaded sockets within the same job.

This is shown in Fig. 6. The y-axis represents average power consumption in Watts over the same x-axis as before (socket IDs). UP indicates uniform power distribution across different sockets of a job while PShifter enforces a non-uniform distribution of the job's power budget, with underloaded sockets (e.g., odd sockets in the 16 socket job) at lower power than overloaded sockets (e.g., even sockets).

Observation 2: PShifter leads to a balanced execution of a parallel job.

As a result of shifting power from underloaded sockets to overloaded sockets, the rate of computation on the underloaded sockets slows down while that on the overloaded sockets speeds up. This leads to a reduction in the computation time on the overloaded sockets and an increase in the computation time on the underloaded sockets, resulting into a balanced parallel execution. The wait times for underloaded sockets are also reduced under PShifter. This can be observed in Fig. 5 by comparing the corresponding UP and PShifter bar plots for 8, 16 and 32 sockets.

Table 1 summarizes the imbalance values for each of the jobs. PShifter reduces the imbalance by 75%, 87%, and 80% for 8, 16, and 32 socket, respectively.

Table 1: Imbalance Reduction

Socket Count	Imbalance with UP	Imbalance with PShifter
8	16%	4%
16	31%	4%
32	49%	10%

Observation 3: PShifter reduces the completion time of the power-constrained parallel job without violating its power budget.

The completion time of a parallel job is constrained by the completion time of the socket with the largest PV value as it hosts the bottleneck process with the longest computation time. Such a socket is on the critical path of parallel execution. Moving power to this overloaded socket speeds up its rate of computation and thus reduces the length of the critical path of parallel execution and the job's completion time. This can be observed in Fig. 5, where PShifter reduces the completion times by 2%, 10%, and 14%, for 8, 16, and 32 socket jobs, respectively.

Observation 4: The power consumption of sockets under PShifter is proportional to the respective computational loads (or computation times under UP).

It is interesting to note that the shape of the computation time curve under UP in Fig. 6 matches the power curve under PShifter for each job. This shows that PShifter shifts just the right amount of power between sockets such that the resulting power consumption of each socket is proportional to its computational load. However, it is important to note that PShifter does not need any prior information about the job's computation time profile. Instead, the decisions about the amount of power to be shifted, the source and the destination of shifted power are made by the local agents and the cluster agent together based on their runtime sensor inputs.

Observation 5: PShifter reduces the energy consumption of the parallel job.

PShifter reduces the energy consumption of 8, 16 and 32 socket jobs by 2%, 14%, and 16%, respectively, over UP. This has two aspects. First, PShifter reduces the completion time of the job without exceeding its power budget. This leads to energy savings. Appropriate shifting of power by PShifter leads to a reduction in wait times of the underloaded sockets. Also the power assigned to the underloaded sockets is lower under PShifter. Hence, while they wait for reduced durations, they consume lower energy even in their waiting phases compared to that in case of UP. This further adds to the energy savings.

Scalability

We evaluated PShifter on up to 3072 cores on 256 sockets. Fig. 7 - 9 compare the performance of miniFE, CoMD, and ParaDiS under UP and PShifter at six different job power budgets for each job size. The x-axis denotes the number of sockets (n) in a job. Average power per socket (Avg_Pow) is indicated at the top of the plot. A job's power budget is set at $n \times Avg\_Pow$. The y-axis represents a job's completion time in seconds. As indicated in the legend, the performance under UP and PShifter is represented by gray and white bars, respectively. The percentages on the top of the PShifter bars indicate performance improvement or reduction in completion time achieved by PShifter over UP. The error bars show the maximum and minimum completion times across five repetitions of each experiment.

PShifter achieves performance improvements of up to 17%, 21%, and 21% for miniFE, CoMD, and ParaDiS, respectively. The geometric mean is 6.5%, 5%, and 7.5% for Fig 7 - 9, respectively. The overhead of running PShifter is no more than 5% (included in the results). In most cases, PShifter causes less run-to-run variation across several repetitions of every experiment compared to UP. This can be observed by comparing the error bars on PShifter and UP in Fig. 7 - 9.

In some exceptional cases, we observe 0% performance gains. For example, consider the 128 socket experiments of miniFE with an average socket power of 55W. PShifter reduces the imbalance by 32% over UP. However, this is countered by the added communication time leading to 0% improvement in performance. Nonetheless, PShifter never loses in performance.

Fig. 11 - 13 compare the energy consumption of miniFE, CoMD, and ParaDiS under UP and PShifter for the six different job power budgets as before. The x-axis is the same as in case of Fig. 7 - 9. The y-axis represents the energy consumption in KiloJoule. The percentages on top of the PShifter bars represent percentage energy savings of PShifter over UP. PShifter achieves energy savings of up to 22%, 16%, and 23% with geometric means of 10.5%, 5.3%, and 9.15%, for miniFE, CoMD, and ParaDiS, respectively. It is important to note here that even in case of 0% performance improvement for the 128 socket job of miniFE at an average socket power of 55W, PShifter achieves 12% energy savings, i.e., when PShifter is at par in performance, it may still gain energy savings over UP. PShifter never loses in energy either.

Observation 6: PShifter scales well with increasing socket count.

Under each power budget (Avg_Pow), PShifter achieves sustainable performance gains even at higher (64, 128, 256) socket counts.

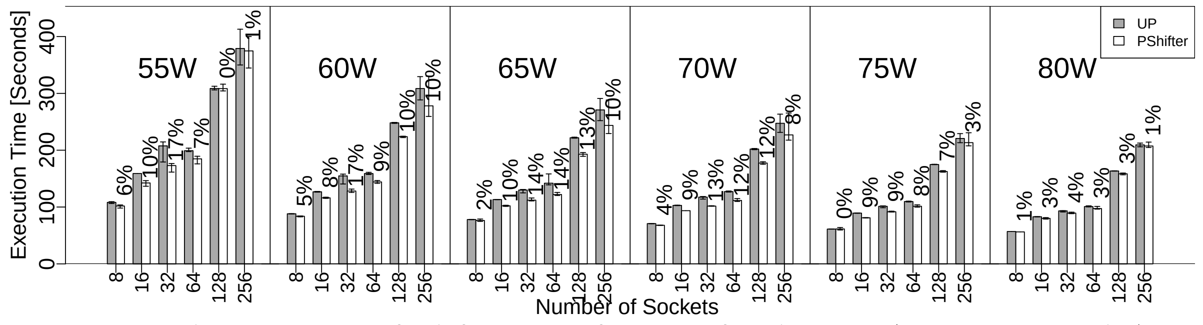

Figure 7: Runtime and % improvement of PShifter over UP for miniFE for job power = (Avg. Power per Socket) × #Sockets

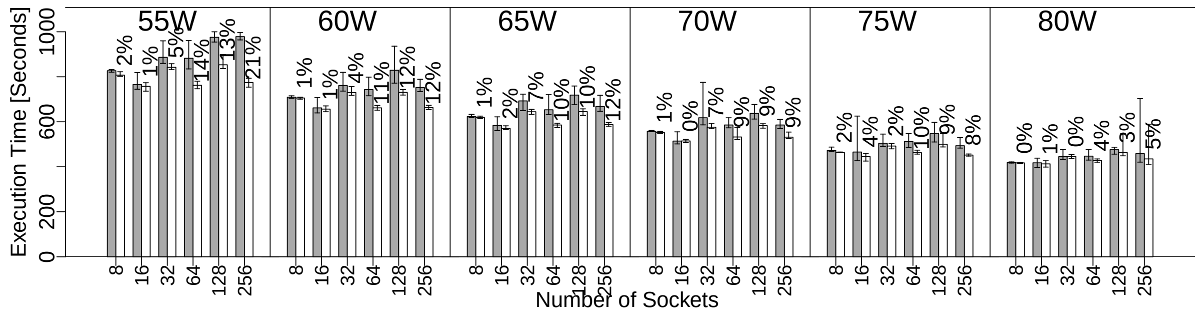

Figure 8: Runtime and % improvement of PShifter over UP for CoMD for job power = (Avg. Power per Socket) × #Sockets

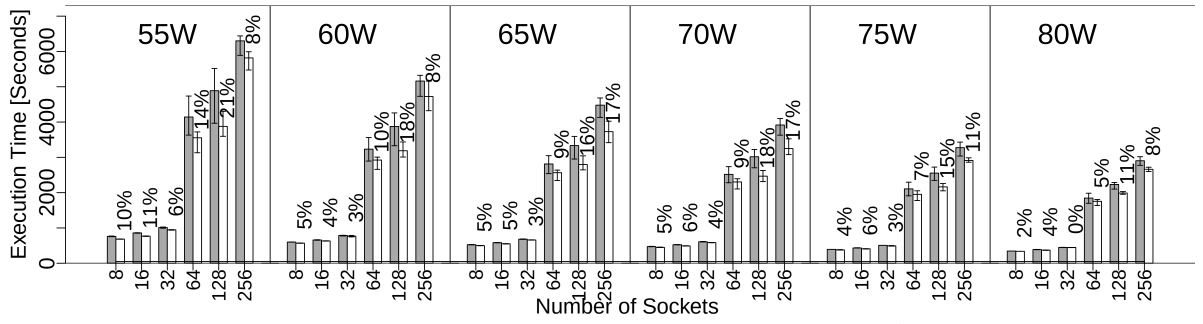

Figure 9: Runtime and % improvement of PShifter over UP for ParaDiS for job power = (Avg. Power per Socket) × #Sockets

Exceptions to this observation are data for miniFE at lowest (55W) and higher (75-80W) power budget. The reasoning for the former case is discussed above. In cases of higher power budgets, it is observed that most of the overloaded sockets are already operating at maximum power (i.e., they are not power constrained) and, hence cannot benefit from additional power. PShifter shifts power away from underloaded to overloaded sockets but these overloaded sockets cannot consume any additional power but underloaded sockets now consume lower power. Hence, the job's energy consumption reduces (5-9%) but performance gains are not significant. This also explains why the performance gains of PShifter at the highest power budget are not as good as in cases of lower power budgets.

Observation 7: PShifter compliments application-specific load balancing and further improves application performance.

Some applications (e.g., ParaDiS) have a built-in application-specific load balancer that moves data from overloaded processes to underloaded ones to reduce the imbalance within a job. This leads to shorter wait times and thus better performance. Our experiments (Fig. 10) indicate when PShifter is combined with the application-specific load balancer, it leads to up to 6% and on an average 3.25%, 4.5% and 1% additional performance improvements on top of load balancing at 60W, 70W, and 80W average power per socket, respectively.

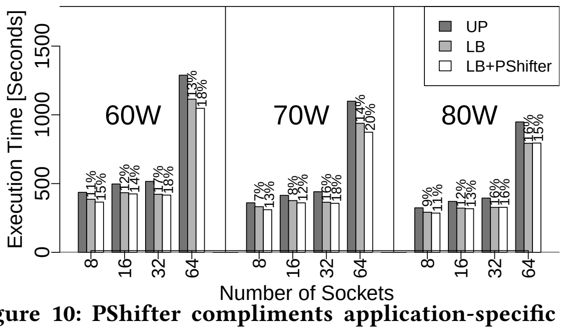

Figure 10: PShifter compliments application-specific load balancer for ParaDiS.

However, not all applications are equipped with an application-specific load balancer and developing one requires domain-specific knowledge and modifications to the application. PShifter avoids this

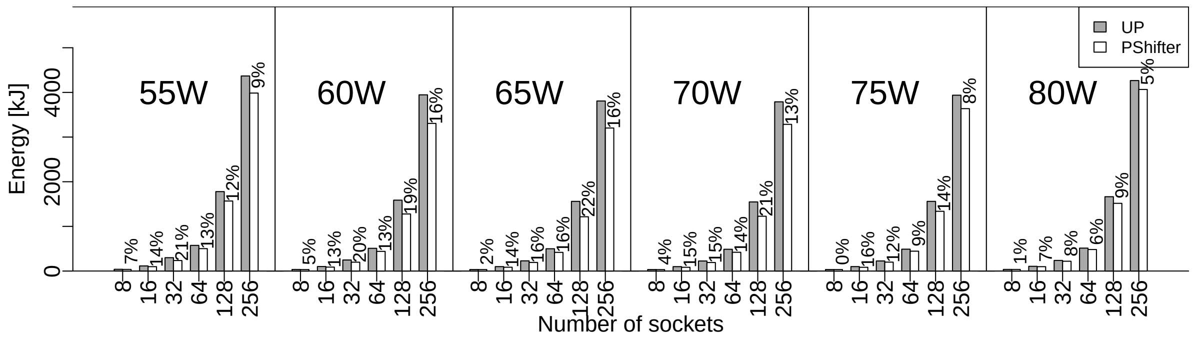

Figure 11: Energy and % improvement of PShifter over UP for miniFE, power budget = (Avg. Power per Socket) × #Sockets

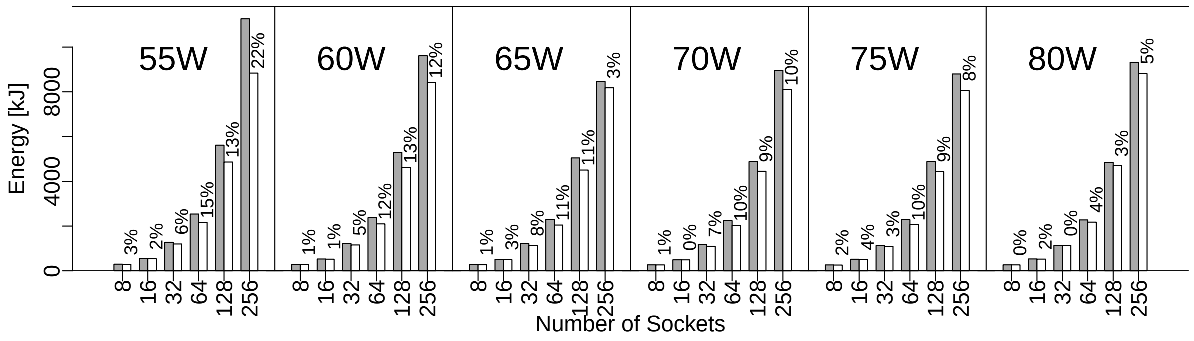

Figure 12: Energy and % improvement of PShifter over UP for CoMD, power budget = (Avg. Power per Socket) × #Sockets

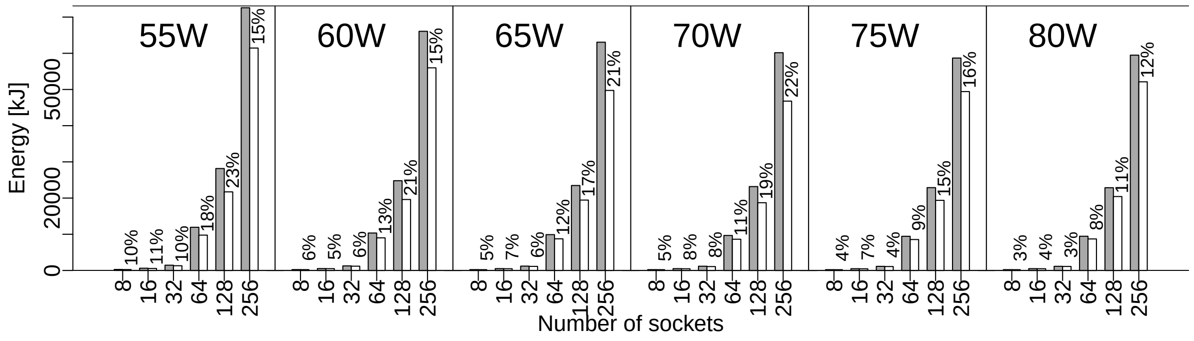

Figure 13: Energy and % improvement of PShifter over UP for ParaDiS, power budget = (Avg. Power per Socket) × #Sockets

developmental effort and provides a general system software solution that can be deployed across applications to reduce workload imbalance by using power as an actuator.

Dynamic Phase-change Detection and Power Management

Observation 8: PShifter detects phase changes at runtime and accordingly shifts power to minimize the new imbalance.

We illustrate the dynamic power management by PShifter in in two figures. Fig. 14 shows the performance imbalance within a 16 socket job for two consecutive phases (shown in the legend) of miniFE. The x-axis represents socket ID and the y-axis represents computation time for each socket per phase without our scheme. The sockets can be grouped into four groups, G1 (socket 1, 3, 5, and 7), G2 (socket 9, 11, 13, and 15), G3 (socket 10, 12, 14, and 16), and G4 (socket 2, 4, 6, and 8) in the increasing order of their computation times in the first phase. In the second phase, loads are reversed between groups, i.e., group G1 becomes the group of overloaded sockets compared to the other groups. G2 and G3 have more or less the same computational load while G4 becomes the group with the least load.

Fig. 15 shows the power profile for the same job as power decisions are made by PShifter during runtime. The x-axis shows the timeline in seconds, the left y-axis shows socket and the right one shows total job power. Power consumed by the 16 sockets of the job and job power are represented by the symbols shown in the legend. The job's power budget is set to 55W*16=880W. All the sockets are initially capped at 55W. As the execution progresses starting from Phase 1, sockets belonging to groups G1 and G2 start giving up power as they are underloaded sockets compared to the rest of the groups while sockets from groups G3 and G4 gradually gain more power in the order of their computational load. After the initial power shifting (from 0 to 7 seconds), socket power stabilizes for each socket until the phase change occurs. After the phase change at 33 secs, power is shifted from groups G2, G3, and G4 to G1 as it is the most overloaded group of sockets. Sockets from groups G2 and G3 stabilize in the range of 40W to 55W while G4 sockets stay at minimum power as they have the least load.

For each curve, you can see multiple data points that show drops in socket power followed by a steep rise in a periodical fashion. The drops are a result of local agent's invocations that lead to lowering of power caps in case of $PV < 0.95$. The rise in power caps is a result of cluster agent invocation that feeds back the unused power into the sockets. The total job power remains consistently below the job's power budget. With PShifter the job completes in 85 seconds.

5.2 Comparison with PTune

In recent work, Gholkar et al. [15] presented *PTune*, a process variation-aware power tuner that uses performance characterization data for all sockets on a cluster and application execution characteristics to minimize the runtime of a job under its power budget. PTune makes static (at the time of job scheduling) decisions for every job about the choice of sockets and the distribution of the job power budget across them. It eliminates inefficient sockets from the job allocation that are unaffordable under an assigned power budget and distributes power non-uniformly across the chosen sockets to counter the effect of performance variation across the cluster. To enforce its policy, PTune requires an application to be *moldable* (number of ranks is modifiable) as the number of nodes in the optimal configuration for a power budget varied based on availability of nodes and the power characteristics of their sockets.

While PTune shares the common objective of achieving performance improvement under a fixed job power budget, it is oblivious of the algorithmic or workload imbalance within a job. As PShifter makes measurement-driven dynamic power decisions at runtime, it detects such workload imbalance. Hence, unlike PTune, PShifter rebalances a job *with or without workload imbalance* on a non-homogeneous machine operating under a power constraint. In addition to this, PShifter easily compliments today's parallel computing system software. It is a stand-alone runtime system that can run alongside every job once it is scheduled by a conventional scheduler. It does not need to co-ordinate in any way with other modules of the system — unlike PTune, which needs to work with the conventional scheduler as it modifies the job allocation for power-constrained jobs. PTune also requires applications to be moldable, which they currently are not.

Table 2 compares the completion times of perfectly balanced miniFE jobs with the two power management schemes, PTune and PShifter, for 8, 16 and, 32 socket jobs at 55W per socket. PShifter achieves an average performance improvement of 9%, 12% and 22% over PTune for 8, 16, and 32 socket jobs, respectively.

Table 2: Completion time of MiniFE with PShifter and with prior work, Power Tuner (PTune)

Socket Count	PTune	PShifter
8	154s	139s
16	174s	153s
32	181s	141s

Unlike PTune, PShifter achieves these speedups without requiring any prior information about the sockets or the application's power-performance curves. It is important to note here that the data (completion time) for PTune does not take the training time (for characterization runs) into account. It only represents the final completion time for the jobs once they are configured by PTune.

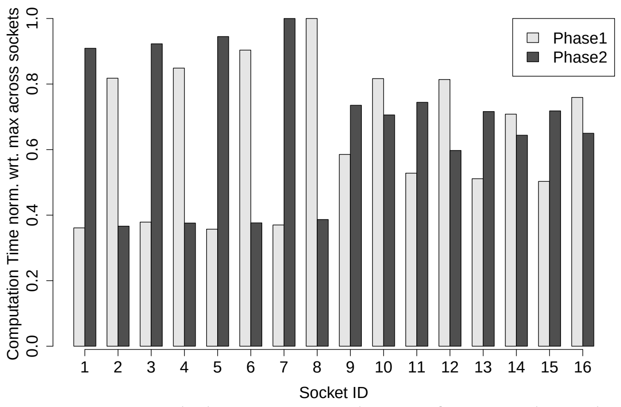

Figure 14: Imbalance in two phases of a 16 socket job

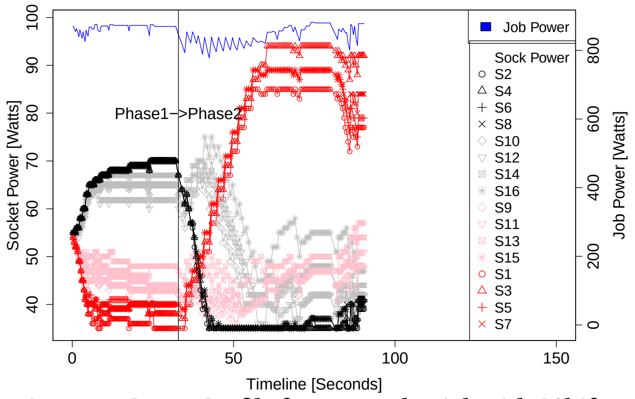

Figure 15: Power Profile for a 16 socket job with PShifter

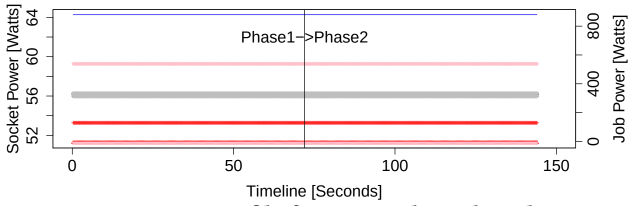

Figure 16: Power Profile for a 16 socket job with PTune

Even though this gives PTune an advantage, PShifter outperforms PTune for balanced codes. Generation of training data would require additional time and energy for PTune that is not accounted for in Table 2.

PTune is oblivious of the runtime imbalances within a job and, hence, is not designed for optimization of load imbalanced jobs unlike PShifter. This is depicted is Fig. 16, which shows the power profile for PTune with static decisions running the same application as shown in Fig. 15 (same axes and power budget as before). At the job initialization, PTune caps sockets non-uniformly by taking the power and performance characteristics of the sockets into account using prior information. This power distribution configuration remains constant throughout the execution. With PTune, this job completes in 144 seconds (40% slower than with PShifter) as the power distribution is not aligned with the load distribution which can only be detected at runtime. The phase change is delayed compared to PShifter as shown in the figure. PTune neither detects nor responds to this phase change as it makes static decisions at job scheduling time unlike PShifter, which makes measurement-driven dynamic decisions.

5.3 Comparison with Conductor

Conductor is a dynamic scheme closest to PShifter with an on-line power-constrained runtime system [26]. Conductor requires applications to be configured with one MPI task per RAPL domain (per processor or socket depending on the Intel chip), and to be OpenMP enabled since it exploits thread parallelism. This hard requirement for hybrid MPI+OpenMP limits Conductor to a subset workloads that excludes MiniFE, which supports MPI only, but not in conjunction with OpenMP. We therefore exclude MiniFE from this comparison.

Conductor speeds up an application's critical path through an adaptive socket power-allocation algorithm that periodically performs dynamic voltage frequency scaling (DVFS) and dynamic concurrency throttling (DCT) based on application behavior and power usage. Conductor consists of two main steps: (a) Configuration space exploration determines the best frequency and thread concurrency level for individual computation tasks under the power limit. (B) Power reallocation intelligently re-assigns power to the application's critical path.

A typical time step in an application may comprise several computational sections. Conductor selects the best configuration (concurrency level and DVFS state) per section. (1) Conductor records power and performance profiles per section for all possible configurations. To reduce the run-time overhead, Conductor performs this step in a distributed fashion by assigning a unique configuration subset to each MPI process and then gathers these profiles at the end of the time step. From these profiles, Conductor creates a list of of power-efficient configurations (that are *Pareto-efficient*) per code section and subsequently selects any new (lower power) configurations during execution.

(2) Conductor monitors power usage per MPI process, estimates the critical path of the application using historical data collected on-line, and reallocates power to speed up the critical path. In more detail, monitoring provides the means to reduce the power consumption on non-critical paths via a low-power configuration that finishes computation just in time without perturbing the critical path. This frees up some power. Differences between MPI processes and paths may be due to an application's load imbalance or differences in power efficiency between sockets. Conductor allocates more power to the processes on the critical path to speed up the application without violating the job power constraint. Conductor performs the power reallocation step at the end of several time steps demarcated using source-level annotations, which must be added by the user.

Fig. 17 and Fig. 18 depict the performance improvement (y-axes) of PShifter and Conductor over Uniform Power for CoMD and ParaDiS, respectively, for different numbers of sockets (x-axes). We used one MPI process with up to 12 threads per socket. For both applications, Conductor consistently performs worse than PShifter. Conductor's lower performance is due to three reasons: (1) Conductor's heuristic for power re-allocation has limitations. The heuristic re-distributes unused job-level power to processors in the order of fraction of time spent near the processor power limit. But this results in inefficient power allocation compared to PShifter's power allocation because for long-running computation Conductor's heuristic depends on *average* power usage, which ignores

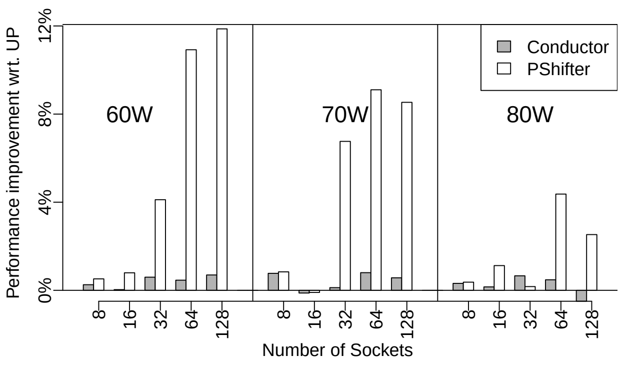

Figure 17: Comparison of PShifter with prior work, Conductor for CoMD.

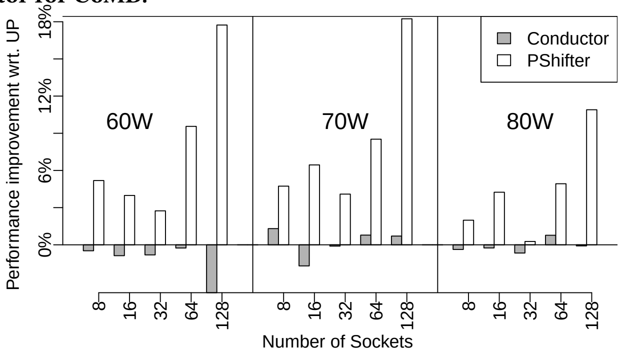

Figure 18: Comparison of PShifter with prior work, Conductor for ParaDiS.

spikes in power demands of the computation tasks. (2) Conductor's configuration selection phase relies on the repetitive nature of code in terms of the computational load and execution paths across processors. Since both the applications were load imbalanced across processors (ParaDiS also being non-deterministic over time), Conductor was forced to perform configuration exploration *sequentially* per process, which contributed to the performance degradation, especially at lower operating frequencies and core counts. Although amortized over a large number of timesteps in our evaluation, this degradation in the configuration exploration step offset the performance gains in subsequent power re-allocation steps. (3) Conductor's heuristic for power management depends on algorithm-level knobs such as number of samples before re-allocating power, fraction of power donated/re-distributed among processors, and the power threshold to trigger power balancing. While it is easy to set up these knobs for repetitive, load-balanced applications, selecting a performance-optimizing combination of them for load-imbalanced applications results in exhaustive search. Our analysis showed that even the best-performing combination of these knobs resulted in thrashing between disjoint power schedules as the load imbalance changed over the run-time for ParaDiS. The performance degradation due to inefficient power schedules offset the performance gains of efficient power schedules. Table 3 summarizes the comparison of PShifter with the closest prior work.

6 RELATED WORK

High performance computing (HPC) has increasingly been driven by power constraints in the past ten years. BlueGene/L was an early system originally based on an embedded processing core to limit power consumption. HPC facilities with DVFS have been studied

Table 3: Comparison of PShifter with PTune and Conductor

Feature	PShifter	PTune	Conductor
Power management	dynamic	static	dynamic
MPI	✓	✓	X
MPI+OpenMP	✓	✓	✓
Non-moldable jobs	✓	X	✓
Load-imbalance	✓	X	✓ low performance
No prior data required	✓	X	✓
Overheads	low	high	high
Deployment needs no modification to status-quo system	✓	X	✓

for a long time. MPI applications were shown to often benefit by trading a slight increase in execution time (or sometimes even none when memory bound) due to running at lower frequencies for a significant reduction in power [13]. DVFS has been combined with performance modeling and prediction to reduce the runtime of MPI codes under energy constraints [40]. A plethora of algorithms exploit DVFS to save energy of HPC jobs [3, 4, 12, 14, 18, 19, 25, 33, 34]. Prior work [6, 24] has also leveraged the effect of concurrency throttling and thread locality to save power and increase performance. While these approaches successfully lower the energy footprint of the jobs, they are ineffective in *enforcing caps* on job-level power budgets. Our work, on the other hand, uses power actuators via Intel's RAPL interface and guarantees that the job-level power constraint is never violated. While RAPL has been explored previously, PShifter is first (to our knowledge) to demonstrate that feedback-driven power reallocation transparently and without any training, results in effective computational load balancing. The novelty of the submission is the combination of the PID controller and RAPL for HPC applications, even across phase changes.

Early work exploited DVFS to reduce CPU frequencies during idle time, e.g., due to early arrival at MPI barriers and collectives [22]. An ILP-based approach to model energy [33] was demonstrated to be effective during the runtime of MPI codes to determine optimal power levels with little to no impact on execution time [34]. Another effective method is to simply power down nodes when not needed to reduce energy [31]. These works were trying to reduce power without sacrificing performance by much when utilizing just one core of a node resulting in underutilization of the system. PShifter differs in that its foremost objective is to guarantee a power constraint followed by trying to *not* degrade performance — and, as experiments showed, successfully so, as most runtimes are *reduced* — while utilizing all cores of a node.

An ILP-based runtime approach has been shown to determine how many cores an application should be run on to stay within a given power budget [23, 41]. Our work differs from this work in terms of granularity and adaptivity. We manage power across resources at processor chip level exploiting dynamically adaptive feedback methods.

Capacity-improving schemes that increase job throughput have been developed under power limitations by exploiting “hardware overprovisioning”, i.e., by deploying more nodes that will be powered at a time [10, 11, 28, 36, 37]. In such a system, the characteristics of a code under strong scaling were used to calculate the optimal number of processors considering core and memory power [38]. Just by exploiting DVFS at the granularity of a job, runtime and power can also be reduced [8, 9]. Modifications to the batch scheduler in how small jobs are backfilled depending on their power profile can further increase job capacity [29]. These schemes do not deal with application imbalance. They, directly or indirectly, assign power budgets to the jobs running on the machine and assume a uniform power distribution within a job. Our solution compliments these approaches by dynamically detecting the imbalance and shifting power within a job to the resources where it is required with the objective of improving the job's performance. We do not explicitly aim at *maximizing* the machine's throughput, which is beyond the scope of this paper, but we often *improve* throughput as a side effect of PShifter. More specifically, each job finishes *early* under our scheme compared to the uniform power distribution scheme. This effectively improves the overall throughput of the system.

In other work, power was shifted within systems via scheduling while adhering to a global power cap [7]. While this approach salvages the unused power, it does not detect wasteful power consumption of the processors at the barriers or other collectives resulting from any imbalance in the job. Our approach is able to reduce this waste and redirect power where it can be better utilized. Power balancing is a technique shown to be able to leverage differences in performance across a set of nodes and their cores [20]. However, one pitfall of this method is its assumption, even though resulting in a good balance, that power and processor frequency have a proportional relationship. Later work indicated that power and processor frequency do not have a linear correlation opening up leverage for more refined power tuning [15]. In a more recent work, waiting cores in the communication phase were power-gated and the saved power was then redirected to other active cores [30]. This work made an assumption that the time to power-gate and wake-up a processor is greater than the communication delay. First, PShifter does not rely on any such assumption. Second, with PShifter, sockets that are not on the critical path operate at lower power not just during communication but also during longer computation phases leading to significant power savings that can be used to accelerate the computation of other sockets on the critical path.

Our PShifter work is unique in that it neither makes any assumptions about the power-performance relationship of the processors nor does it require prior information about the variation across the processors. This, and the fact that it adapts dynamically to changes in execution behavior, are the biggest virtues of our scheme. PShifter makes measurement-based decisions by monitoring the state of the system. As this is a dynamic runtime system, it can also sense the imbalance resulting from the unequal division of work across processors in iterations in addition to the static imbalance induced due to performance variation across processors under power caps. Proposed frameworks like Redfish, the PowerAPI, and Intel's GEOPM [1, 2, 16] can integrate PShifter as a unique closed-loop feedback-based policy for job power management on a cluster. PShifter also relieves the application developers of the burden to explicitly indicate phase changes as required by the APIs (like GEOPM) as PShifter automatically detects phases without any explicit information from the developer.

7 SUMMARY

We presented PShifter, a feedback-based hierarchical solution for managing power of a job on a power-constrained system. PShifter makes dynamic decisions at runtime solely based on measurements. Unlike prior work, it does not depend on any a prior data about the application or the processors. At job level, PShifter employs a cluster agent that opportunistically improves the performance of a job while operating strictly under its power constraint. It does so by dynamically re-directing power to where it is needed. At processor level, PShifter employs local agents that aim to reduce the energy of the processors they manage. They achieve this by reducing the power of the processors that incur long wait times. Our evaluations show that PShifter achieves a performance improvement of up to 21% and energy savings of up to 23% compared to a naïve approach. Compared to a static power scheme, PShifter improves performance by up to 40% and 22% for codes with and without phase changes, respectively. Compared to a dynamic power scheme, it improves performance by up to 19%. PShifter transparently and automatically applies power capping non-uniformly across nodes in a dynamic manner adapting to changes during execution, simply by linking the PShifter library with or preloading it to an application.

8 ACKNOWLEDGEMENTS

We extend our thanks to Livermore Computing for their support. This work was funded in part by subcontracts from Lawrence Livermore and Lawrence Berkeley National Laboratories, NCSU's Consortium for Advanced Simulation of Light Water Reactors (CASL) hub funded by DOE, and NSF grants 1217748, 0958311 and 1058779. Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-750271).

REFERENCES

[1] Panel on power api/redfish/geopm. https://eehpcwg.llnl.gov/documents/conference/sc16/SC16_Laros_Eastep_Benson_API.pdf.

[2] Redfish api. http:https://www.dmtf.org/standards/redfish/.

[3] P. E. Bailey, A. Marathe, D. K. Lowenthal, B. Rountree, and M. Schulz. Finding the limits of power-constrained application performance. In *SC*, 2015.

[4] S. Bhalachandra, A. Porterfield, S. L. Olivier, and J. F. Prins. An adaptive core-specific runtime for energy efficiency. In *IPDPS*, pages 947–956, May 2017.

[5] V. Bulatov, W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis. Scalable line dynamics in paradis. In *Supercomputing*, 2004.

[6] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Supercomputing*, pages 157–166, 2006.

[7] D. A. Ellsworth, A. D. Malony, B. Rountree, and M. Schulz. Pow: System-wide dynamic reallocation of limited power in hpc. In *High-Performance Parallel and Distributed Computing*, pages 145–148, 2015.

[8] M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Optimizing Job Performance Under a Given Power Constraint in HPC Centers. In *Green Computing Conference*, pages 257–267, 2010.

[9] M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Utilization driven power-aware parallel job scheduling. *Computer Science - R&D*, 25(3-4):207–216, 2010.

[10] M. E. Femal and V. Freeh. Boosting data center performance through non-uniform power allocation. In *International Conference on Autonomic Computing*, 2005.

[11] M. E. Femal and V. W. Freeh. Safe overprovisioning: using power limits to increase aggregate throughput. In *In International Conference on Power-Aware Computer Systems*, December 2005.

[12] V. Freeh, F. Pan, N. Kappiah, and D. K. Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *PPoPP*, pages 164–173, June 2005.

[13] V. Freeh, F. Pan, N. Kappiah, D. K. Lowenthal, and R. Springer. Exploring the energy-time tradeoff in mpi programs on a power-scalable cluster. In *IPDPS*, 2005.

[14] R. Ge, X. Feng, W. c. Feng, and K. W. Cameron. Cpu miser: A performance-directed, run-time system for power-aware clusters. In *2007 International Conference on Parallel Processing (ICPP 2007)*, pages 18–18, Sept 2007.

[15] N. Gholkar, F. Mueller, and B. Rountree. Power tuning hpc jobs on power-constrained systems. In *PACT*. ACM, 2016.

[16] R. E. Grant, M. Levenhagen, S. L. Olivier, D. DeBonis, K. T. Pedretti, and J. H. L. III. Standardizing power monitoring and control at exascale. *Computer*, 49(10):38–46, Oct 2016.

[17] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[18] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *IEEE Transactions on Computers*, 56(4):444–458, April 2007.

[19] C.-h. Hsu and W.-c. Feng. A power-aware run-time system for high-performance computing. In *Supercomputing*, page 1, 2005.

[20] Y. Inadomi, T. Patki, K. Inoue, M. Aoyagi, B. Rountree, M. Schulz, D. Lowenthal, Y. Wada, K. Fukazawa, M. Ueda, M. Kondo, and I. Miyoshi. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *SC*, pages 78:1–78:12, 2015.

[21] Intel. Intel-64 and IA-32 Architectures Software Developer's Manual, Volumes 3A and 3B: System Programming Guide. 2011.

[22] N. Kappiah, V. Freeh, and D. K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. In *SC*, Nov 2005.

[23] A. Langer, E. Totoni, U. S. Palekar, and L. V. Kalé. Energy-efficient computing for hpc workloads on heterogeneous manycore chips. In *Proceedings of Programming Models and Applications on Multicores and Manycores*. ACM, 2015.

[24] D. Li, B. R. de Supinski, M. Schulz, K. W. Cameron, and D. S. Nikolopoulos. Hybrid mpi/openmp power-aware computing. In *IPDPS*, volume 10, pages 1–12, 2010.

[25] M. Y. Lim, V. Freeh, and D. K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs. In *SC*, 2006.

[26] A. Marathe, P. E. Bailey, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski. A run-time system for power-constrained hpc applications. In J. M. Kunkel and T. Ludwig, editors, *High Performance Computing*, pages 394–408, Cham, 2015. Springer International Publishing.

[27] S. Mintchev and V. Getov. Pmpi: High-level message passing in fortran77 and c. In B. Hertzberger and P. Sloot, editors, *High-Performance Computing and Networking*, pages 601–614, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[28] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski. Exploring Hardware Overprovisioning in Power-constrained, High Performance Computing. In *International Conference on Supercomputing*, pages 173–182, 2013.

[29] T. Patki, D. K. Lowenthal, A. Sasidharan, M. Maiterth, B. Rountree, M. Schulz, and B. R. de Supinski. Practical Resource Management in Power-Constrained, High Performance Computing. In *HPDC*, 2015.

[30] L. Piga, I. Paul, and W. Huang. Performance boosting opportunities under communication imbalance in power-constrained hpc clusters. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 31–40, Aug 2016.

[31] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Workshop on compilers and operating systems for low power*, 2001.

[32] B. Rountree, D. H. Ahn, B. R. de Supinski, D. K. Lowenthal, and M. Schulz. Beyond DVFS: A First Look at Performance under a Hardware-Enforced Power Bound. In *IPDPS Workshops*, pages 947–953. IEEE Computer Society, 2012.

[33] B. Rountree, D. K. Lowenthal, S. Funk, V. Freeh, B. R. de Supinski, and M. Schulz. Bounding energy consumption in large-scale mpi programs. In *SC*, Nov 2007.

[34] B. Rountree, D. K. Lowenthal, M. Schulz, V. Freeh, and T. Bletsch. Adagio: Making dvs practical for complex hpc applications. In *ICS*, Nov 2009.

[35] Sandia National Laboratory. Mantevo project home page. https://software.sandia.gov/mantevo, June 2011.

[36] O. Sarood. *Optimizing Performance Under Thermal and Power Constraints for HPC Data Centers.* PhD thesis, University of Illinois, Urbana-Champaign, Dec 2013.

[37] O. Sarood, A. Langer, A. Gupta, and L. V. Kale. Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *SC*, 2014.

[38] O. Sarood, A. Langer, L. V. Kale, B. Rountree, and B. de Supinski. Optimizing Power Allocation to CPU and Memory Subsystems in Overprovisioned HPC Systems. In *Proceedings of IEEE Cluster 2013*, Sept 2013.

[39] K. Shoga, B. Rountree, M. Schulz, and J. Shafer. Whitelisting msrs with msr-safe. In *3rd Workshop on Extreme-Scale Programming Tools at SC*, Nov. 2014.

[40] R. Springer, D. K. Lowenthal, B. Rountree, , and V. Freeh. Minimizing execution time in mpi programs on an energy-constrained,power-scalable cluster. In *PPoPP*, May 2006.

[41] E. Totoni, A. Langer, J. Torrellas, and L. Kale. Scheduling for hpc systems with process variation heterogeneity. In *Technical Report YCS-2009-443, Department of Computer Science, University of York*, 2014.