Power Tuning for HPC Jobs under Manufacturing Variations

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

ABSTRACT

As we approach the exascale era, power has become a primary bottleneck. The US Department of Energy has set a power constraint of 20MW on each exascale machine. To be able achieve one exaflop in 20MW, it is necessary that we use power intelligently to maximize performance under a power constraint.

Most production-level parallel applications that run on a supercomputer are tightly-coupled parallel applications. A naive approach of enforcing a power constraint for a parallel job would be to divide the job's power budget uniformly across all the processors. However, previous work has shown that a power capped job suffers from performance variation of the processors due to manufacturing variations leading to overall sub-optimal performance. We propose a 2-level hierarchical variation-aware approach of managing power at machine-level. At macro-level, *PPartition* partitions machine's power budget across jobs to assign a power budget to each job running on the system such that the machine never exceeds its power budget. At micro-level, *PTune* makes job-centric decisions by taking the performance variation into account. For every moldable job, it determines the optimal number of processors, the selection of processors and the distribution of the job's power budget across them, with the goal of maximizing the job's performance under its power budget.

Our evaluations show that at micro-level, PTune achieves a performance improvement of up to 29% compared to the naive approach. PTune does not lead to any performance degradation, yet frees up almost 40% of the processors for the same performance as that of the naïve approach, under a hard power bound. PPartition is able to achieve a throughput improvement of 5-35% compared to uniform power distribution.

1. INTRODUCTION

The supercomputing community is headed toward the era of exascale computing, which is slated to begin around 2020. Today's fastest supercomputer, Tianhe-2, consumes 17.8MW to deliver 33.86PFlops [**?**]. If we were to build an exascale machine with today's technology it would consume up to 350MW of power. A typical power plant generates 1 GWatt of power, which is sufficient to power 700,000 homes [**?**]. The US DOE has set a power constraint of 20MW per future exascale systems to maintain a feasible electrical power demand. In order to get an exaflop under this constraint, we need at least an order of magnitude improvement in power efficiency with respect to today's system [**?**, **?**, **?**, **?**].

Exascale systems are expected to be power-constrained. One of the key contributions in the power-constrained domain is hardware overprovisioning[**?**]. The idea is to procure more hardware capacity than what can be operated at its maximum power under a power constraint. In other words, the power that is budgeted for a such a system is not sufficient to run the entire system at maximum performance.

Fig. **??** depicts this idea to lay the foundation for the rest of this paper. Let the hardware overprovisioned system consist of X processors and let the power budgeted for this system be Y Watts. As shown in the figure, with Y Watts total system power only a part of the system (< Y processors) can be utilized at peak power (collection of nodes in red). Another valid configuration is to utilize the entire system at low power. One of the several other intermediate configurations is to use medium power levels and utilize a portion of the system larger than that at peak power but smaller than that at low power. Depending on the application's characteristics (memory-boundedness, compute-boundedness, communication-boundedness), different applications achieve optimal performance on different configurations. In a nutshell, power procured for a system must be managed as a malleable resource to maximize performance of an overprovisioned system under a power constraint.

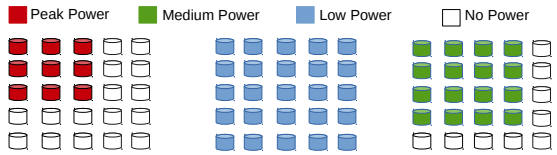

Figure 1: Hardware Overprovisioning on a Power-Constrained System

To facilitate power management, the semiconductor industry and other hardware manufacturers are providing var-

ious hardware features like power clamping and on-chip power measurement mechanisms [**?**], as well as component power measurement infrastructures [**?**, **?**]. Several mechanisms of modulating the power levels of processors are available. Dynamic Frequency Voltage Scaling (DVFS) allows the programmer to set the frequencies and voltages of the processors at different levels leading to different power levels. From Sandy Bridge onwards, processors Intel support) Running Average Power Limit (RAPL) interface [**?**] that allows the programmer to bound the power consumption of the package (PKG) and the memory (DRAM). A package is a single multi-core processor shipped by Intel. In this work, we use the PKG domain of RAPL to bound the power consumption of processors.

Different processors within the same model and stepping consume different amounts of power to complete the same work, i.e., the processors are not equally power efficient. This variation can be attributed to the manufacturing variability introduced by the fabrication process. Under uniform power bounds, the variability in power efficiency of the processors translates into variation in their performance. This observation has been documented in [**?**].

Let *power efficiency* be defined as the performance per Watt. Performance is quantified in terms of number of instructions retired per second (IPS). We show that under a power bound the manufacturing variability transforms into variation in peak power efficiency. We make two important observations. First, not all processors are equally power efficient at a power bound. The most efficient processors are most efficient at lower power bounds whereas the least efficient processors are the most efficient at higher power bounds.

To be able to achieve the goal of maximizing science under a power constraint, at machine-level (macro-level) job schedulers need to be power-aware, i.e., in addition to job scheduling decisions, they also need to make power scheduling decisions across jobs assigning power budgets to the jobs. Most of the jobs on supercomputers are comprised of multiple and often coupled parallel scientific simulations that execute on several processors simultaneously. A naïve approach of enforcing a power constraint for a job is to evenly distribute power across all the processors of the job. When a job is assigned a power budget PB and it is scheduled on n processors, each of the processors will be bounded at $\frac{PB}{n}$ as per this näive strategy. We call this *uniform power*. A job allocation may consist of processors that are not equally power efficient at a single power bound, i.e., capping them at uniform power would not lead to optimal performance. Hence, at micro-level, i.e., within a job, it is necessary to take the variation in power efficiency into account to make optimal power decisions.

We propose a 2-level hierarchical solution that performs Power Partitioning (PPartition) at macro-level and Power Tuning (PTune) at micro-level to maximize the science done per Watt. PPartition aims at improving the throughput of the machine by (re-)partitioning a machine's power budget across jobs while scheduling them. PTune is a variation-aware power manager for moldable jobs whose number of processors can be chosen at dispatch time. For every job, PTune maximizes its performance under its job power budget (assigned by PPartition) by determining the following:

- The optimal selection of a subset of processors from the available ones; and
- the distribution of the job's power budget across the selected processors.

PTune follows an off-line approach that makes these decisions before the beginning of job execution.

PTune achieves a job performance improvement of up to 29% over uniform power. PTune does not lead to any performance degradation, yet frees up 40% of the resources compared to uniform power. PPartition and PTune together improve the throughput of the machine by 5-35% compared to conventional scheduling on an overprovisioned machine.

In this work, we make the following contributions:

- We characterize the performance of contemporary Intel processors with multiple codes at multiple power bounds. Performance variations of up to 30% are observed across these processors. Using this data we conduct a power efficiency study that forms the basis for the work.
- We propose *PTune*, a variation-aware job power tuner that optimizes a job for performance under a strict job power constraint.
- We propose *PPartition*, a variation-aware resource manager that manages a machine's power budget as a resource to maximize the throughput of the machine under a machine-level power constraint.
- We evaluate PTune and PPartition using three of the NAS Parallel benchmark codes [**?**] and a molecular dynamics proxy application CoMD [**?**].

The paper is organized as follows. Section **??** presents the motivation for this work. Section **??** states the problem statement. Section **??** gives an overview of our approach. Sections **??** and **??** describe per-job power tuning and cross-job power partitioning, respectively. Section **??** discusses the implementation of PTune. Section **??** presents the experimental setup and the evaluation of the model. Section **??** discusses related work. Section **??** summarizes the contributions.

2. MOTIVATION

Until recently, the research community has focused on minimizing energy usage of supercomputers. Considering the US DOE mandate for a power constraint per exascale site, efforts need to be directed towards minimizing the wasteful usage of power while maximizing performance under this constraint.

As stated in the previous section, uniform power capping is the naïve approach of enforcing a job level power constraint. In order to understand what happens under such a scheme, we characterized the performance of 600 Ivy Bridge processors on a cluster called Catalyst at Lawrence Livermore National Laboratory, CA. We ran three of the NAS Parallel Benchmark (NPB) suite codes [**?**], viz., Embarrassingly Parallel (EP), Block Tri-diagonal solver (BT) and Scalar Penta-tridiagonal solver (SP), and CoMD, a molecular dynamics proxy application from the Mantevo suite [**?**] at several different processor power bounds on all the processors. The processor power bounds were set using the PKG domain of RAPL. The results are depicted in Fig. **??**. The x-axis represents power in Watts while the y-axis represents

Instructions Retired per Second (IPS) in billions. We observed that the cluster becomes non-uniform under power bounds. Performance variations of up to 30% are observed across this cluster for these applications. Maximum performance of 77, 50, 80, and 60 billions IPS is achieved for CoMD, EP, BT and SP, respectively. This variability is due to the manufacturing variations that manifests itself in fluctuations in unbounded power consumption across different processors.

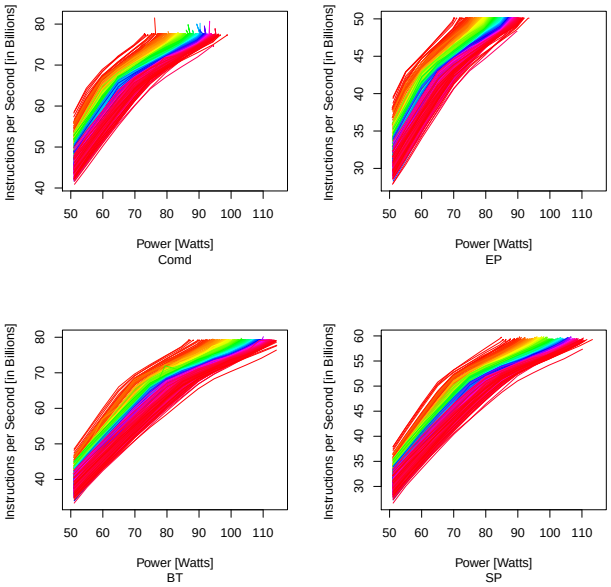

Figure 2: IPS vs. Power for each processor on Catalyst. The rainbow color palette is used to represent several processors. In each plot, the curves in red (bottom) depicts the least efficient processors while the curves in orange (top) represent the most efficient processors.

On studying these results further, we found that this variability in performance translates into variation in peak *power efficiency* of the processors.

Power Efficiency

Let *power efficiency* be defined as the number of instructions retired per second per Watt of operating power.

Fig. **??** represents the power efficiency curves of the processors on the cluster. We run several benchmarks (NPB EP, BT, SP) and a proxy application (CoMD) on the cluster at 15 power levels from 51W to 120W in intervals of 5W. The x-axis represents the operating power in Watts and the y-axis represents the power efficiency in billion IPS/W. The rainbow palette used in the previous graph is reused to represent different processors. Each curve (or each color) in the plots corresponds to a unique processor.

We make the following observations from these experiments:

- The power efficiency of a processor varies with its operating power and is non-monotonic. It is also workload-dependent.
- Peak power efficiency varies across processors. Manufacturing variability translates into variation in peak power efficiency of the processors.
- Efficient processors are most efficient at lower power bounds whereas the inefficient processors are most efficient at higher power bounds.

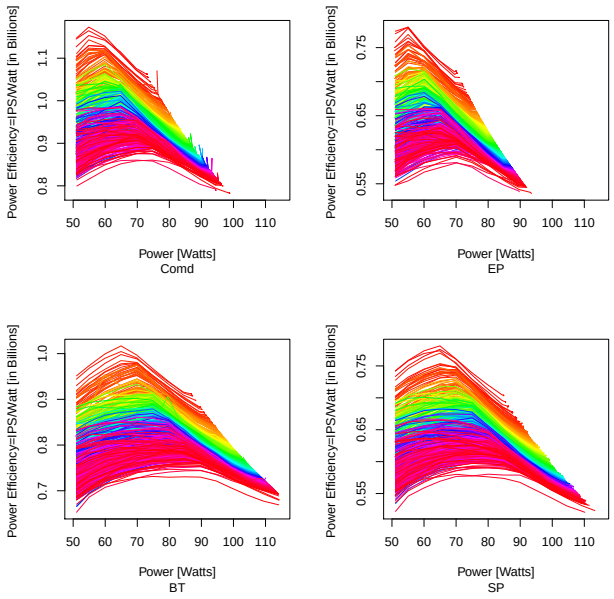

Figure 3: Power Efficiency in IPS/W vs. Operating power. The rainbow color palette is used to represent several processors. In each plot, the curves in red (bottom) depict the least efficient processors while the curves in orange (top) represent the most efficient processors.

The bottom line is that there is a unique local maximum in every power efficiency curve that occurs at disparate power levels for different processors. Starting from the minimum power, increasing the power assigned to a processor leads to increasing gains in IPS. However, increasing the power beyond the peak efficiency point of a processor leads to diminishing returns. Hence, when power is limited, processors should operate at power levels close to their peak efficiency to maximize the overall efficiency of the system. Since the peak efficiency points for efficient processors are at lower power levels than for the inefficient processors, the optimal configuration should select lower power levels for efficient processors and higher power levels for inefficient processors to maximize performance. On the contrary, a naïve / *uniform power* scheme caps all the processors at identical power bounds. Hence, it is sub-optimal.

An optimal algorithm should aim at leveraging the non-uniformity of the cluster to maximize the performance of a job under its power constraint. We propose *PTune*, a power-performance variation-aware power tuner that exactly does this for each job. For every job, given a power budget, it determines the following: (1) the optimal number of processors (say N); (2) selection of N processors; and (3) the power distribution (say pi) across the selected N processors.

3. PROBLEM STATEMENT

The problem statement is stated as follows:Given a machine level power budget, how should the machine's power be distributed across (a) jobs and (b) tasks within jobs on a given system, where (b) is discussed later. For (a), the process of making these decisions at macro-level of jobs is called *power partitioning*. Each job on the machine receives its own power partition.

We address the following questions:

1. How many partitions do we have at a time? I.e., we need to determine how many jobs should be scheduled at a time.
2. What is the size of each of the power partitions? I.e., we need to determine the power budget assigned to each of the jobs.

For (b), at the micro-level, given a hard job-level power budget P_{Ji}, we need to determine the optimal number of processors, n_{opt}, with a power distribution (p_1, p_2, ... , $p_{(n_{opt}-1)}$, $p_{n_{opt}}$) such that performance of the job is maximized under its power budget. The constraint on the power distribution is expressed as

$$\sum_{k=1}^{n} p_k \leq P_{Ji}; min\_power \leq p_k \leq max\_power_k.$$

Here, min_power is the minimum power that needs to be assigned to a processor for reliable performance and max_power_k is the maximum power consumed by the k^{th} processor (uncapped power consumption) for an application. The performance of a job can be quantified in terms of number of instructions retired per second (IPS). For a parallel application on n processors, the effective IPS is the aggregated IPS over n processors ($JobIPS_n$). Hence, the objective function is

$$Maximize(JobIPS_n). \quad (1)$$

A processor's IPS is a non-linear function of the power at which it operates. Each processor can be power bounded at several levels using the RAPL interface and thus forced to operate at various power levels within a fixed range. We know that unbounded power consumption is variable across processors while achieving the same unbounded (peak) performance. This is depicted in Fig. **??**. The x-axis is the power at which the processor operates and the y-axis is the IPS (in billions) of the processor. Each solid curve corresponds to the most efficient processor while the dotted curve correspond to the least efficient processor. The following two observations are made from this data:

1. On a single processor, the performance (IPS) achieved at any fixed power level is different for different workloads.
2. The performance of an application on two different processors at any fixed power level is not the same.

This means that when determining the optimal distribution of power across processors it is necessary to take the processor characteristics and the application characteristics into account. One solution may not fit all applications. The optimal configuration for an application on one set of processors may be different from that on another set of processors because of manufacturing variation.

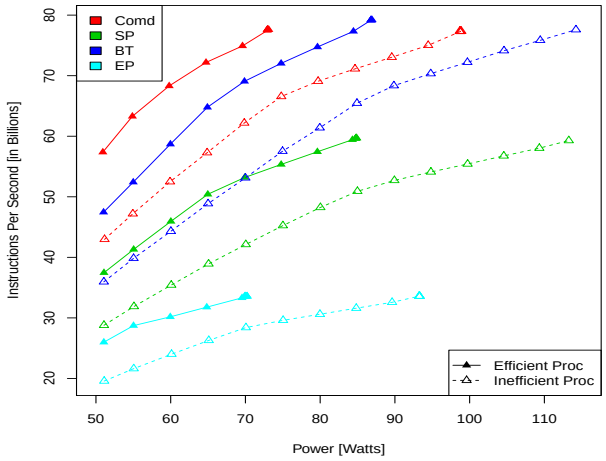

Figure 4: IPS vs. Power for efficient and inefficient processors.

4. PROPOSED SOLUTION

We propose a 2-level hierarchical approach of managing power as a resource (see Fig. **??**). The parameters of the model are given in Table **??**. We make the assumption that the power consumption of the interconnect is zero, i.e., interconnect power is beyond the scope, and so are task-to-node mapping effects on power. We only consider processor power in this work and assume moldable jobs.

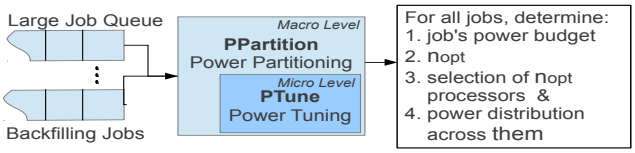

Figure 5: Hierarchical Power Manager.

At the macro-level, we propose *PPartition*, a technique of partitioning a machine's power budget across jobs while scheduling them. Once a job is dispatched by a conventional scheduler (e.g., slurm or Maui/pbs), PPartition calculates its power budget. If the required power is not available, it steals power from the previously scheduled jobs and provisions this power for the new job. If sufficient power cannot be made obtained, PPartition overrides the conventional scheduler's decision based on free resources (nodes) and does not schedule this job until required power is available.

At the micro-level, we propose *PTune*, a power balancing model that determines the distribution of a job's power budget (one job at a time) across an optimal selection of processors (among all free resources) to maximize the performance of a job under its power budget.

5. PTUNE

Fig. **??** depicts the micro-level power tuner. For each job Ji, power budget P_{Ji} is calculated at the macro-level by the Power Partitioner. For every job with this assigned power budget,

PTune answers the following questions:

Table 1: Model Parameters

Parameter	Description	PPartition	PTune
N_{max}	maximum number of processors on a machine	Input	N/A
N_{alloc}	number of processors already alloced to jobs	Output	N/A
$P_{m/c}$	power budget of the machine	Input	N/A
n_{req}	number of processors requested by a job	Input	Input
n_{opt}	optimal number of processors for a job	Output	Output
n	number of processors for a job under its power budget	N/A	Variable
P_{Ji}	power budget of the i^{th} job	Output	Input
p_k	power cap of k^{th} processor within a job	N/A	Output
min_power	minimum processor power cap	Input	Input
max_power_j	maximum processor power cap of the j^{th} processors	Input	Input
power-ips table	characterisation data Table **??**	Input	Input

Table 2: Power vs. IPS Table

Power Cap[W]	IPS in Billions	Measure Power [W]
60	46.43	59.99
80	64.83	79.88
100	76.33	99.43
120	79.13	104.66

1. How many (n_{opt}) and which processors should a job run on?
2. What should be the power ($p_1, ..., p_{n_{opt}}$) assigned to each of the n_{opt} processors?

Let us start by addressing the first question. A processor needs to be assigned at least the minimum power (min_power) that is architecturally defined for every family of processors and is constant across all processors, i.e., the lower bound on the thermal design power (TDP). However, the maximum or unbounded power (max_power_k) consumption of the processors is *non-uniform* due to manufacturing variations [**?**].

Fig. **??** shows the maximum power consumption of 600 Ivy Bridge processors. The x-axis represents all the processor sorted by their power consumption and the y-axis represents the maximum power consumption in Watts. In order to maximize the performance of a job under a constant power budget, it is necessary to choose more efficient processors from the set of available processors.

5.1 Sort the Processors

The first step towards determining the optimal configuration is to sort the available processors by their relative power efficiency. This is equivalent to sorting them by their unbounded power consumption. Let the sorted set of processors be indexed by k.

We divide this distribution of processors into quartiles, viz., Q1, Q2, Q3 and Q4, in the order of efficiency and pick processors from one or more of these quartiles for evaluation purposes.

5.2 Bounds on Number of Processors

The lower bound on n, $n_\perp$, can be calculated by determining the maximum number of processors that can be capped at their maximum power, max_power_k, under the power budget. The selection of processors is reformed in the sorted order as described above. $n_\perp$ is given by the largest value of n that satisfies the following constraint:

$$P_{Ji} \geq \sum_{k=1}^{n} max\_power_k.$$

The upper bound on n, $n_\top$, represents the number of processors that can be operated at min_power under the power budget. The bound $n_\top$ is calculated as follows:

$$n_\top = \frac{P_{Ji}}{min\_power}.$$

The processor count, n, is iterated from $n_\perp$ to $n_\top$, and in each step, the next efficient processor is added to the set of processors. Job-level performance, $JobIPS_n$, is calculated in each iteration by $DistributePower()$ for the power budget P_{Ji} and a given number of processors, n.

$$JobIPS_n = DistributePower(n, P_{Ji}, (p_1, ..., p_{n-1})) \quad where n_\perp \leq n \leq n_\top, \quad (2)$$

The optimal number of processors, i.e., n_{opt}, is the value of n at which a job's IPS is maximized. PTune leads to $n_{opt} \leq n$. Thus, PTune tends to reduce the number of processors required for a moldable job so that spare processors can be utilized by other jobs.

$$JobIPS_{n_{opt}} = max(JobIPS_{n_\perp}, JobIPS_{(n_\perp+1)}, ..., JobIPS_{n_\top}). \quad (3)$$

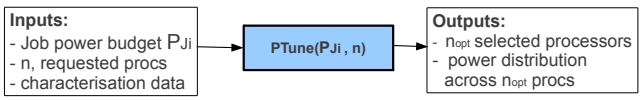

Figure 6: PTune

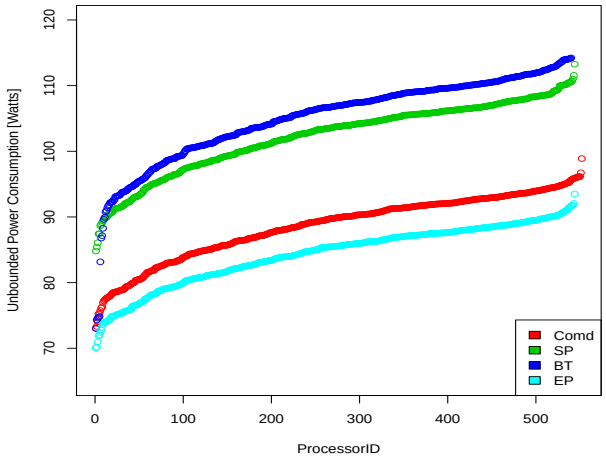

Figure 7: IPS vs. Power for efficient and inefficient processors.

5.3 Distribute Power : Mathematical Model

DistributePower(), takes three inputs, viz., the number of processors n, the job's power budget P_{Ji}, and the power distribution across $n-1$ processors determined in the previous iteration. The output of this function is the maximum job IPS that can be achieved under P_{Ji} Watts with n processors. It also calculates the optimal power caps, $(p_1, ..., p_n)$, for n processors, which forms an input for the next iteration. This can be mathematically expressed as follows:

$$DistributePower(n, PB, (p_1, ..., p_n)) = DistributePower((n-1), PB - p_n, p_1, ..., p_{(n-1)}) + getProcIPS(n, p_n) \quad (4)$$

getProcIPS(k,p_k) accesses the power-ips characterization data gathered statically. It returns the expected performance (IPS) of the k^{th} processor when it is capped at p_k Watts.

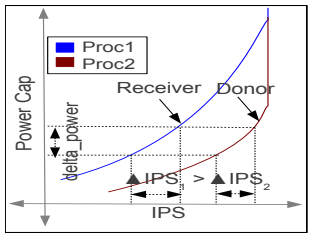

Figure 8: Identifying the donor and the receiver of discrete power

5.4 Power Stealing and Shifting

An iteration consists of two main phases, viz. Power Stealing and Power Shifting. *DistributePower*() consists of two main steps:

(1) Power is stolen in discrete quantities (*delta_power*) from the $n-1$ processors to provision power for the n^{th} processor (see Fig. **??**). The victim/donor processor is the one that suffers minimum loss in IPS when *delta_power* is stolen from it. If aggregate stolen power is at least *min_power*, an additional n^{th} processor is added to the processor set.

(2) Power is shifted from a donor to a receiver in discrete quantities, *delta_power*, across the n processors. The victim/donor processor is identified by (1). The receiver is the processor that gains maximum IPS on receiving *delta_power*.

6. PPARTITION

Fig. **??** depicts the macro-level power partitioning algorithm. The power partitioner co-operates with the conventional scheduler. PPartition receives information on the performance variations across processors. It always chooses the most efficient n_{req} processors of the available processors (or a subset thereof) to schedule a job. When job J_i is dispatched by the conventional scheduler, its initial power budget, P_{Ji}, is calculated as shown in the diagram. If the required power and processing resources are available, PTune determines the optimal configuration for the job and the job is scheduled. If resources are available but the required power is insufficient, power is stolen from already scheduled jobs. This is called power repartitioning (lower right blue/shaded box in Fig. **??**) and detailed next.

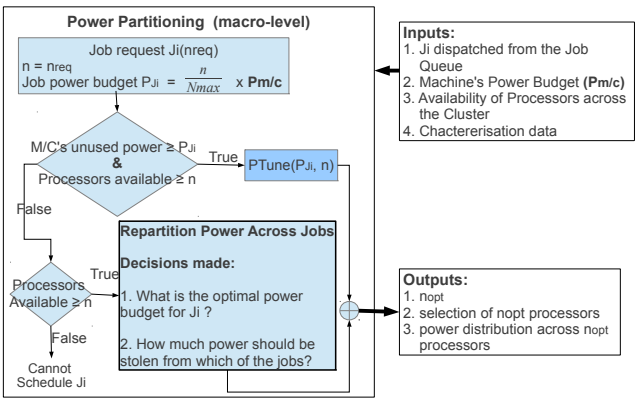

Figure 9: PPartitioning: Repartitioning Power.

Power Repartitioning

The power repartitioning algorithm is shown in Fig. **??**. As all of the machine power budget is already used up by the $N_{allocated}$ processors, a fair power share for the new job is calculated as

$$P_{Ji} = P_{m/c} * \frac{n}{n + N_{allocated}}, \quad (5)$$

where n=n_{req}.

Algorithm : Repartitioning Power for accomodating the ith job

```
n = nreq
/* Job's power budget proportional to the proportion
   of machine requested */
PJi = Pm/c  x n/Nmax

// Get nopt for the job under power budget PJi.
nopt=PTune(PJi, n)
While (nopt < n) {
      n = nopt
      // PJi proportional to the proportion of busy
      // processors requested
      PJi = Pm/c  x n/(Nalloc+n)
      // Recompute nopt for the reduced power
      nopt=PTune(PJi, n)
}

/* Repartition power across jobs to provision power
   for the i^th job.*/
for( k in 1 to (i-1)) {
      power_to_be_stolen[k] =  PJi  X jobpowerbudgets[i] / Σ jobpowerbudgets
      total_stolen_power = total_stolen_power +
                  ShrinkPartition(power_to_be_stolen[k], k)
}
If ( total_stolen_power <PJi){
      PJi = total_stolen_power
      // Recompute nopt
      nopt=PTune(PJi, n)
}
```

Figure 10: PPartition, PTune and the interaction between them.

The job is initially power tuned for the requested n_{req} processors under P_{Ji} Watts. If there are sufficient processors left for the required power budget, i.e., $n_{opt} \leq n$, then n is set to n_{opt}. Notice that this initial PTune call may have reduced the number of processors from n_{req} to a lower number n_{opt}. Due to that, we need to recompute (in the

while loop) the proportionate power the new job with n_{opt} processors should have due to power partitioning across all jobs. This new power budget, P_{Ji}, then becomes the base for another PTune, and so on, until the number of processors for the new job reaches a fixed point (stabilizes) in the while loop. The fixed point guarantees a fair power level relative to other jobs, but we still need to find other jobs to steal just enough power for this job.

In the following for loop, power is stolen from each of the scheduled jobs in a proportionate manner to each others' power budget. This is accomplished by *partition shrinking*, which consists of (1) stealing just enough power and (2) power tuning for the remaining power of a job and the same number of processors (since we assume moldable but not malleable applications). Here, we steal as much power as possible while retaining heterogeneous power bounds across a job's processors to respect process variations and thus ensure a high IPS.

The aggregate stolen power of other jobs is offered to the new job. If the stolen power is less than that of the last PTune call, which was P_{Ji}, then the new job needs to be tuned one more time. If the stolen power was sufficient for this last tuning step, the new job is scheduled and existing ones are power re-tuned under the new settings. If, however, the stolen power is insufficient, no power is redistributed, i.e., all jobs remain unchanged in their power settings and the new job is deferred until at least another job completes.

7. IMPLEMENTATION

We modified the libmsr [**?**] library to gather the processor characterization data. We implemented a power-performance profiler using the MPI profiling interface (PMPI) that invoked various subroutines of the libmsr library to assess the power and the performance of MPI applications. We captured several fixed counter values, power consumption and completion times for each application on all the processors. The processor power consumption was measured using Intel's RAPL interface. This characterization data is made available available to PTune and PPartition.

We assume that the jobs are moldable. Our power manager works in co-ordination with the conventional job scheduler. Once a job is dispatched by the conventional scheduler, the power manager (PPartition+PTune) determines its power budget, the selection of processors from those available, and the power distribution (or processor power caps) across them. We simulated the conventional job scheduler in R. We assume a large job queue ($>$ 384 processes) and a backfilling queue ($<$ 48 processes). We assume up to Nmax=550 nodes with 12 cores each (6600 processes). If the power manager decides to schedule the job, power distribution across its processors (and power repartitioning if required) is enforced using RAPL.

For evaluating our power manager, we compare it with a uniform power manager that distributes $P_{m/c}$ evenly across processors. We assume three different configurations, processors capped at *min_power*, 75W, and the upper Thermal Design Power (TDP), which is the maximum power. Jobs are scheduled on a portion of the machine constrained by a power limit of $P_{m/c}$.

8. EXPERIMENTAL SETUP

Experiments were conducted on the *Catalyst* cluster at Lawrence Livermore National Laboratory (LLNL). It is a 324-node Ivy Bridge cluster. Each node has two 12-core Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz processors and 128 GB of memory. We used MVAPICH2 version 1.7. The codes were compiled with the Intel compiler version 12.1. The msr-safe kernel module provides direct access to Intel RAPL registers via libmsr [**?**]. We used the package (PKG) domain of RAPL that provided us the capability of capping power for each of the processors in an experiment. The environment was simulated in R.

We again used EP, BT, and SP from the NPB suite and CoMD from the Mantevo suite in their pure MPI versions. We exponentially increase the node count for our experiments. The inputs were weakly scaled for different node counts. We report performance in terms of completion time in seconds and power in Watt. The reported numbers are averages across ten runs.

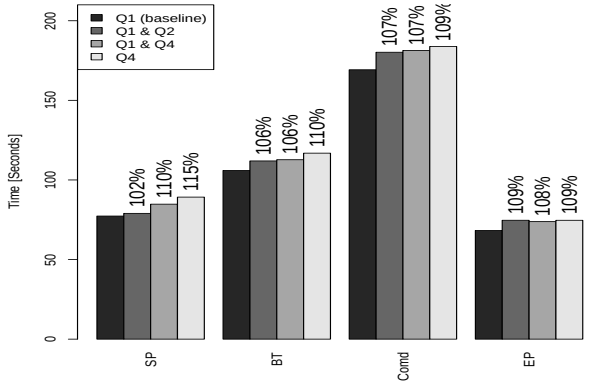

Figure 11: Performance variation on 16 processors

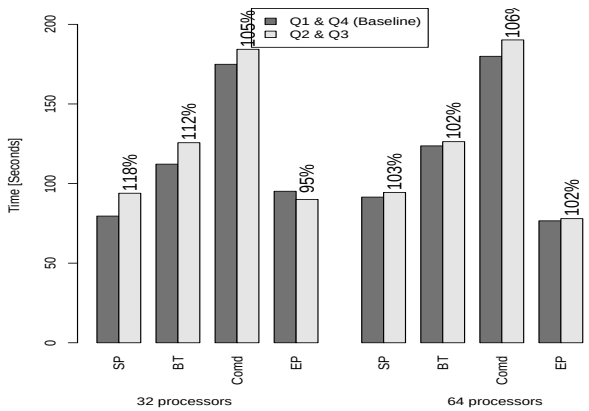

Figure 12: Performance variation on 16 and 32 processors

9. RESULTS

Experiments were conducted for single job power tuning and multi-job power partitioning.

Process Variation — Sorting is a Must

Process variation or (CMOS process variation) is the variation in the attributes of the transistor that occurs at the time of fabrication. This causes variability in the unbounded power consumption of the processor chips, which is translated into variation in performance under a power constraint. Previous work [**?**] and Section **??** has already established that the cluster is inhomogeneous under a power constraint because of this process variation. We also observe that scheduling a job on different sets of fixed number of processors under a constant power budget leads to variation in the performance of a parallel job.

We present a selection of configurations to demonstrate this behavior in Figures **??** and **??**. The x-axis represents the codes and the number of processors. The y-axis indicates the completion time in seconds. The codes are run on several combinations of processors from one or more quartiles of the processor distribution. The numbers on the top of the bars indicate percentage slowdown with respect to the baseline. The processors are uniformly capped at 51W in this set of experiments, i.e., they maintain a constant job power budget of 8KW, 16KW, and 32KW for 16, 32, and 64 processor experiments, respectively. The baseline for 16 processor experiments (Fig. **??**) is the performance on the processors belonging to quartile Q1. For 32 and 64 processors (Fig. **??**), the baseline is the performance on the processors belonging to Q1 and Q4 (also see legends). Q1 consists of the most efficient processors whereas Q4 consists of the least efficient processors. We observe performance slowdown ranging from 2% to 18%.

We observe that performance deteriorates as we include less efficient processors (Q2, Q3, Q4) in the mix. Hence, the optimal selection of n_{opt} processors should consist of the most efficient processors from the available ones. At macro-level, PPartition chooses efficient processors to schedule the dispatched job, and at micro-level, PTune aims at further eliminating the inefficient ones from this selection.

PTune

Let us evaluate the effectiveness of PTune using the aforementioned codes. In Fig. **??**, we present results for three different combinations of processors belonging to different quartiles. There are three data points corresponding to each code.

In the Figure, n_{LOWER} (synonymous with $n_{\perp}$) is the minimum number of processors that operate at maximum power such that their aggregate power does not violate the job level power constraint. This configuration most closely resembles the worst case power provisioning as processors are not power constrained. PTune is the data point corresponding to optimal configuration suggested by the power tuner. Uniform power corresponds to the naïve approach of distributing the job's power budget evenly across all processors in a job. This is the baseline configuration.

Performance

Fig. **??** represents on the y-axis performance (top graph) in terms of wall-clock time in seconds and the number of processors recommended by the power manager (bottom graph) over different codes and quartiles to which the processors belong (x-axis). The numbers on the bars indicate the runtime reduction and utilized number of processors relative the baseline in percent.

We observe a performance improvement of up to 22%. The gains are dependent on the combination of processors from different quartiles as well as on workload. PTune is able to free up to 38% of the resources while achieving similar or higher performance than the baseline configuration.

Scalability

We evaluate PTune on up to 128 processors. Fig. **??** presents results addressing the scalability of PTune. PTune achieves performance improvements of as much as 29% with a minimum of 1%. More significantly, in case of the minimal performance improvement, PTune frees up 23% of the processors, which subsequently become available to the next scheduled jobs.

We observe an error of less than 2% between the total job power consumption (measured via RAPL) of the PTune recommended configurations and the assigned job-level power budget across all experiments.

PPartition

In this section, we perform a macro-level evaluation of our 2-level model. We simulate the conventional scheduler that dispatches jobs from multiple queues, one at a time. Let n be the number of processes. The scheduler handles 3 queues, 1 large job queue ($n \geq 768$ or $n \geq 64$ processors), and two backfill queues ($n \leq 48$ or $n \leq 4$ processors, $48 \leq n \leq 384$ or $4 \leq n \leq 32$ processors. Larger jobs are scheduled first followed by backfilling jobs to improve the system utilization. We assume $Nmax = 550$ processors. Our job mix consists of 25% jobs from each EP, SP, BT and CoMD.

We assume a hardware overprovisioned machine with machine power budget $P_{m/c} = 28KW$. Fig. **??** depicts a scenario in which the job scheduler is oblivious of power management. The machine's power budget is uniformly distributed across all the processors. We call this naïve scheduling. The scheduler schedules job on this machine as long as the required number of processors are available. Fig. **??** depicts the scenario when our power manager is in action. It schedules jobs in a variation-aware and a power-aware manner. The x-axes in both the plots represent job identifiers ordered by the time that they are dispatched under the conventional scheduler. We can see that the large 64 processor job is scheduled first followed by the backfilling jobs. The left y-axis denotes job performance in IPS. Each of the red, green and blue curves represents a job's performance as more and more jobs are scheduled over time (moving right along the x-axis).

Our scheduler starts with jobs at high power budget and, hence, high performance. But as more jobs are dispatched, power is stolen from the previously scheduled jobs. This leads to a drop in their performance. In return, we are able to schedule more jobs at the expense of the performance of already running jobs. In this scenario, PPartition is able to schedule 58 jobs whereas the power-oblivious scheduler is able to schedule only 36 jobs. The performance of most of the first 36 jobs (that are scheduled under both the schemes) of our approach is at least as good as the naïve one. In addition to these jobs, our power control is able to schedule 22 more backfill jobs that further improve the overall throughput of the machine (compared to the naïve approach) under the same power constraint.

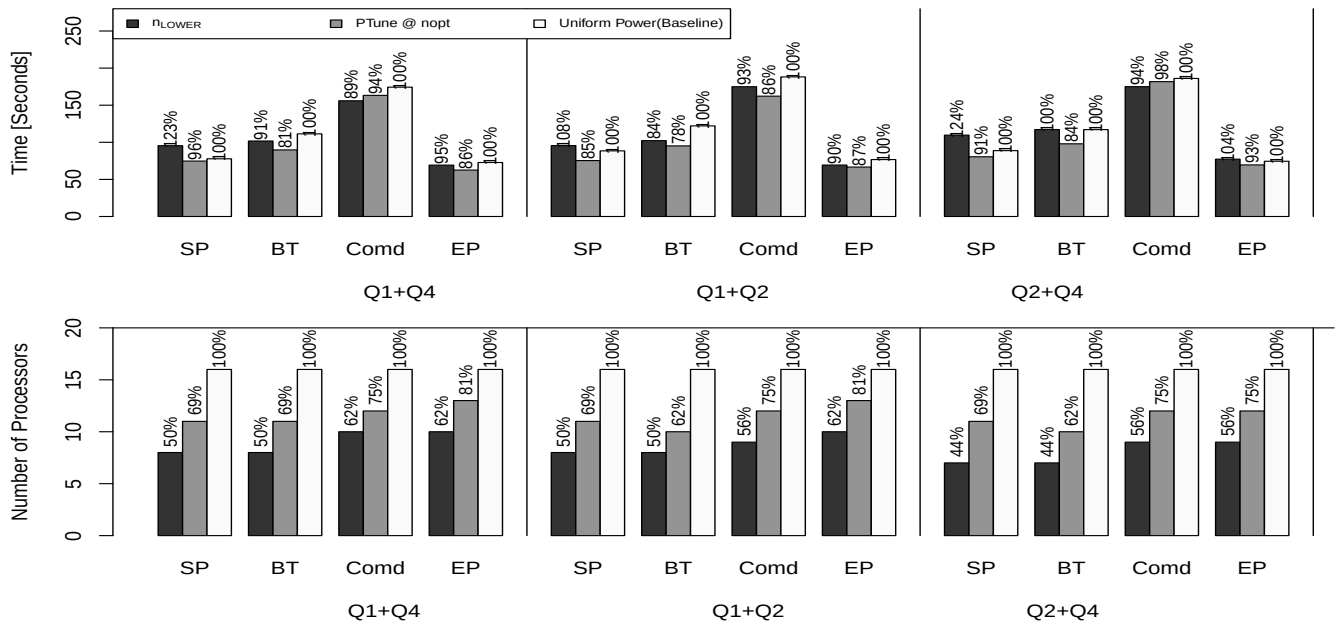

Figure 13: Evaluation of PTune on 16 processors from one or more quartiles.

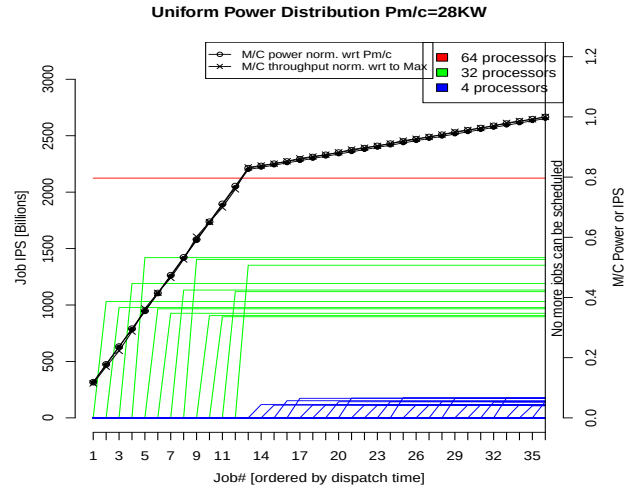

Figure 15: Uniform power distributed across the machine

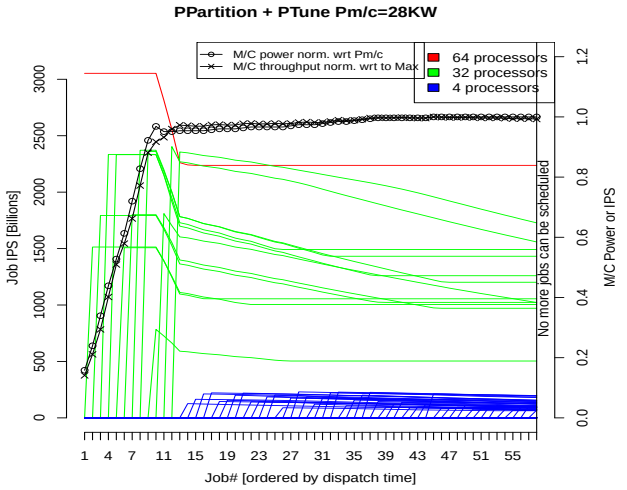

Figure 16: PPartition + PTune

The right y-axes depict the job-level power as a fraction of (normalized to) the overall provisioned system power, $P_{m/c}$, in one line graph (circles) and normalized to maximum power in the other (crosses). Both graphs track each other closely, but under our power control, the machine power is reached much earlier after about 10 jobs whereas all 36 jobs are required to reach this level in the naïve case. These initial jobs are also able to achieve higher performance under our scheme (>1500 Billions IPS) than that in the naïve case (900 to 1500 Billion IPS and 2100 Billion IPS for the large job) before the rest of jobs are scheduled, and these jobs would thus terminate earlier as they have progressed further under our power control than for the naïve case. This shows that when there are fewer jobs running on a machine, our power manager is able to direct all machine power to where the work is to maximize performance under a power constraint unlike the naïve approach.

Fig. **??** presents a comparison of the throughput of our scheme compared to three other naïve schemes. The x-axis denotes the machine's power budget and the y-axis depicts the throughput of the machine normalized to a maximum throughput of 39KW (left set of bars). Uniform capping schemes assume that an appropriate number of randomly selected processors on the machine are already capped at $P_{m/c}/N_{max}$, 75W (mid-way between minimum power and TDP), and TDP, such that their aggregate power does not exceed the machine's power budget. The rest of the processors in these configurations are not available to the con-

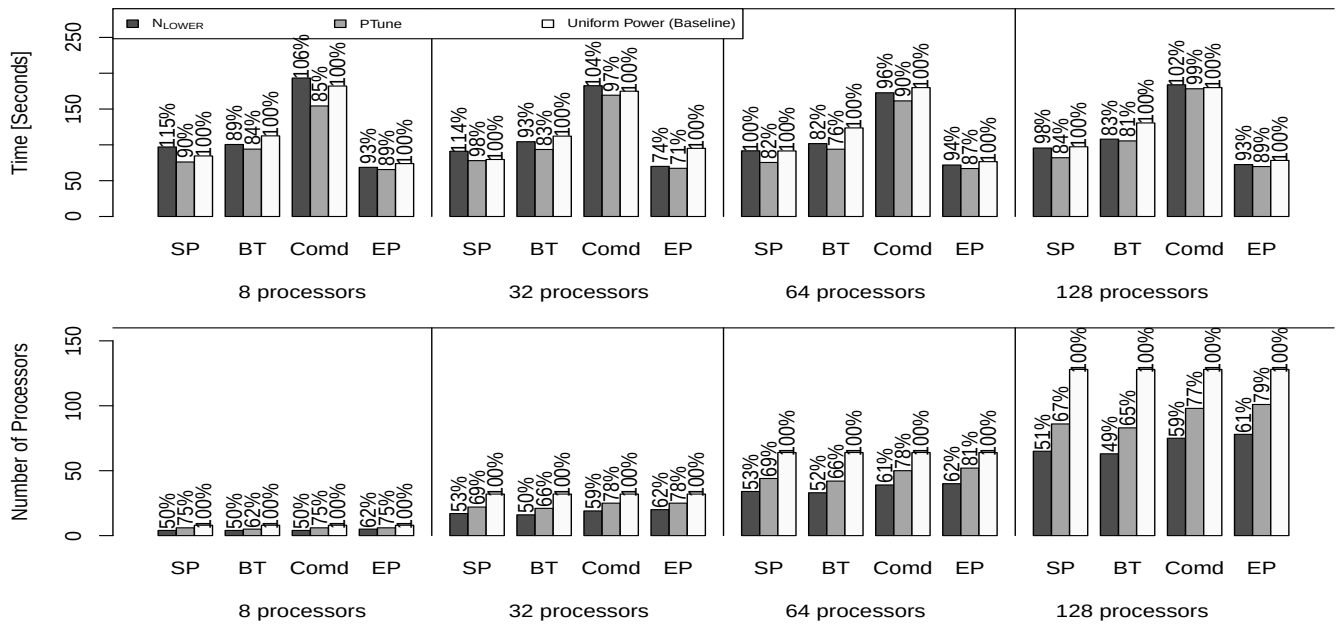

Figure 14: Evaluation of PTune on processors from Q1 and Q4 quartiles.

ventional scheduler in the naïve scheme. PTune+PPartition represents our model that makes variation-aware decisions about scheduling jobs across the entire machine under a machine-level power constraint. The percentages on the top of the bars indicate how much lower the throughput per naive scheme is compared to our solution. Our model consistently achieves 5-35% higher throughput.

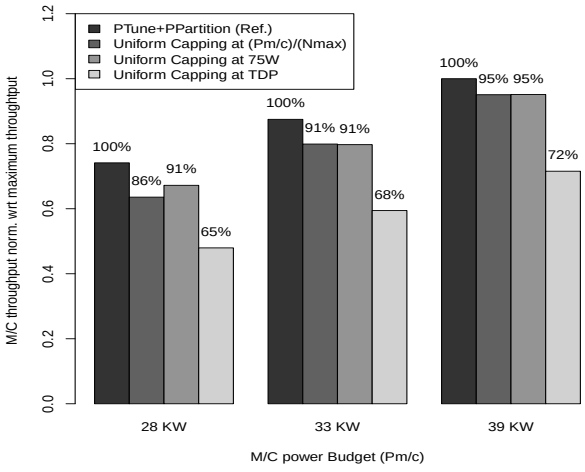

Figure 17: Throughput

Fig. **??** depicts the performances of all the jobs that are scheduled under each scheme indicated along the x-axis. The y-axis denotes job's performance normalized wrt. to the aggregate job performance under respective schemes. We see that our approach (scheme 1) is able to schedule a much larger number of jobs than the naïve scheduling (scheme 2) by trading off performance of some jobs.

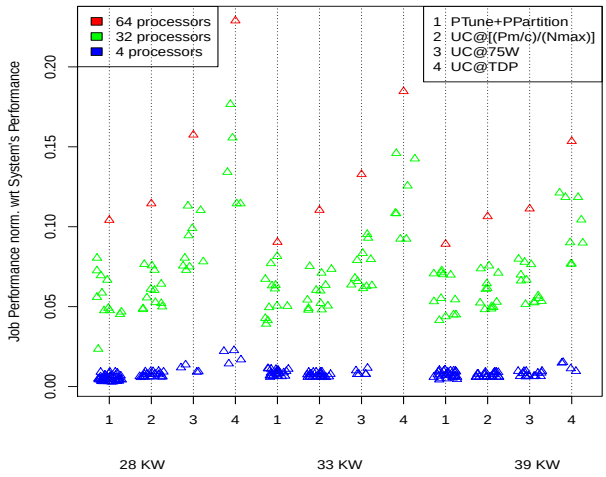

Figure 18: Job performance. A job is represented by a triangle.

10. RELATED WORK

Energy has been an important issue in high performance computing (HPC) for over a decade. Supercomputers as old as BlueGene/L have been built with the goal of maximizing power efficiency. Power-scalable clusters that are equipped with voltage and frequency scaling have existed for over a decade that enabled researchers to study the energy problem in HPC. Freeh et al. [**?**] investigated the energy-time trade off of MPI applications to prove that it is feasible to save energy by scaling the processor down to lower energy levels with or without time penalty depending on the application. Springer et al. [**?**] proposed a combined approach of

performance modeling and performance prediction for minimizing the execution times of MPI applications under energy bounds. They used voltage and frequency scaling on single cores of a small cluster of up to 10 nodes for their experiments. In addition, there is abundant work presenting algorithms that use frequency and voltage scaling mechanisms for energy savings [?, ?, ?, ?, ?]. In contrast, our work uses power clamping via the Intel RAPL interface like Rountree et al. [?]. Totoni et al. [?, ?] presented an ILP based runtime system that schedules work on one or more cores of a multi-core chip to meet the power or performance constraint. Our work differs from this work in terms of granularity. We manage resources at processor chip level. We use either all or no cores of a chip or a multi-core processor.

System-wide solutions for power constraint systems have been proposed that aim at increasing the throughput of systems by leveraging the idea of *hardware overprovisioning* [?, ?, ?, ?, ?]. Sarood et al. [?] proposed a scheme of determining an optimal number of nodes under strong scaling of applications executing on an overprovisioned system while distributing power between CPU and memory. Etinski et al. [?, ?] proposed the use of DVFS at job scheduling-level to save energy and improve overall job performance. Patki et al. [?] proposed power-aware backfilling to improve the throughput of the system. Ellsworth et al. [?] presented a power scheduler that enforced a system-wide power bound by reallocating power across the cluster. Our work differs from all of the above because our approach is variation-aware. It takes the performance variation into account while scheduling and tuning jobs for performance.

The two foundational papers for performance optimization across inhomogeneous processors are [?] and [?]. The former is the first mention of processor inhomogeneity under a power bound in the HPC context. The latter provides a much more detailed analysis of the phenomenon across multiple clusters and provides a set of simple algorithms for intelligent power balancing. These algorithms, while groundbreaking, suffered from two serious limitation. First, the processor power model assumed CPU clock frequency increased proportionally with power. While that is a useful simplification, our work here shows that the story is not nearly so simple.

Second, the algorithms assumed the ideal number of nodes to use was fixed a priori. Making this assumption reduces the problem to something far more tractable, but results in making direct comparisons between our approach and their's nearly impossible. In this paper, we have tackled what we consider to be a far more difficult problem: determining the ideal number of nodes from first principles. For the same node count, we would expect similar performance from both our approach and Inadomi's. Highlighting the benefit of our first-principles approach by comparing to Inadomi running on a non-ideal number of nodes violates an assumption of Inadomi's algorithm. While the result of both of our approaches is a power schedule, we are solving a fundamentally different problem.

Kappiah et al. [?] presented a system that saves energy at the expense of execution time by scaling down the frequencies of the cores when they encounter slack time in an MPI application. Rountree et al. [?] used linear programming to establish a bound on optimal energy savings of an MPI application and presented a runtime algorithm to save energy in HPC applications with negligible delay [?]. Power conservation by means of turning off unwanted nodes is proposed in [?]. In the above presented solutions, authors used one core per node and their goal was to maximize energy savings with minimal impact on the execution time. In contrast to these solutions, we are intolerant to performance degradation. We use multicore processors and our goal is to minimize the completion time as long as we stay within the power budget.

11. SUMMARY

We presented a hierarchical variation-aware machine-wide solution for managing power on a hardware overprovisioned machine. It consists of a macro-level Power Partitioner that makes power and job scheduling decisions and a micro-level Power Tuner that determines the optimal processor selection and their power caps for a job, such that its performance is maximized under a power constraint. PTune achieves up to 29% improvement in performance as compared to uniform power capping. It does not lead to any performance degradation, yet frees up to 40% of resources as compared to uniform power capping. PPartition is able to improve the throughput of the machine by 5-35% compared to naïve sheduling under the same machine power budget.

We established that under a power constraint, the variability in performance transforms into variation in peak power efficiency. We believe that this variation in power efficiency should be one of the primary considerations in the future power management research.