Optimizing Power Allocation to CPU and Memory Subsystems in Overprovisioned HPC Systems

Osman Sarood, Akhil Langer, Laxmikant Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL
{sarood1, alanger, kale}@illinois.edu

Barry Rountree, Bronis de Supinski
Lawrence Livermore National Laboratory
{rountree4, bronis}@llnl.gov

***Abstract*—Energy consumption and power draw pose two major challenges to the HPC community for designing larger systems. Present day HPC systems consume as much as 10MW of electricity and this is fast becoming a bottleneck. Although energy bills will significantly increase with machine size, power consumption is a hard constraint that must be addressed. Intel's Running Average Power Limit (RAPL) toolkit is a recent feature that enables power capping of CPU and memory subsystems on modern hardware. In this paper, we use RAPL to evaluate the possibility of improving execution time efficiency of an application by capping power while adding more nodes. We profile the strong scaling of an application using different power caps for both CPU and memory subsystems. Our proposed interpolation scheme uses an application profile to optimize the number of nodes and the distribution of power between CPU and memory subsystems to minimize execution time under a strict power budget. We validate these estimates by running experiments on a 20-node (120 cores) Sandy Bridge cluster. Our experimental results closely match the model estimates and show speedups greater than 1.47X for all applications compared to not capping CPU and memory power. We demonstrate that the quality of solution that our interpolation scheme provides matches very closely to results obtained via exhaustive profiling.**

I. Introduction

In the past, researchers have primarily focused on reducing energy consumption by using Dynamic Voltage and Frequency Scaling (DVFS). Applications do not yield a proportional improvement in performance as the processor frequency is increased [**?**]. This is mainly because memory accesses are much slower compared to processor frequency. Memory accesses therefore introduce stalls in the processor cycles. The extent of improvement in application performance resulting from increased processor frequency depends on the application's computational and memory demands. As we approach the exascale era, the thrust is more on power consumption than on energy minimization. A strict power constraint poses a hard research challenge. DOE has currently set a bound of 20MW for an exascale system, therefore available power must be used efficiently to achieve the exascale goal. Scaling frequency via DVFS does not guarantee a strict limit on the power consumption of a processor. However, the recently released Intel's Sandy Bridge family of processors provide an enticing option of limiting the power consumption of a processor chip and memory (also available in IBM Power6 [**?**], Power7 [**?**] and AMD Bulldozer [**?**] architectures). The power consumption for package and memory subsystems can be user-controlled through the RAPL (Running Average Power Limit) library [**?**].

In this work, we use Intel's *power_gov* [**?**] library that in turn uses RAPL to cap power of memory and package subsystems in order to optimize application performance under a strict power budget for an *overprovisioned* system. An overprovisioned system [**?**] has more nodes than a conventional system operating under the same power budget. It can not simultaneously power all nodes at peak power. However, capping package (CPU) and memory power below peak power can enable an overprovisioned system to operate all nodes simultaneously. Under a strict power budget, running the application on fewer nodes with a higher CPU/memory power per-node can sometimes be less efficient than running it over larger number of nodes with relatively lower power per node. Capping the CPU and memory power to lower values enables us to utilize more nodes for executing an application. However, each additional node utilized has a fixed cost of powering up the motherboard, power supply, fans and disks referred to as the *base* power. The base power of a node determines the ease with which additional nodes can be utilized in an overprovisioned system. The opportunity cost of base power for these additional nodes is the performance benefit that can be achieved by increasing the CPU and memory power for the existing set of nodes. This opportunity cost can vary between applications.

To the best of our knowledge, this work proposes the first extensive experimental study that *optimizes* number of nodes and the subsequent distribution of power between CPU & memory for an application under a strict power budget. The major contributions of this work are listed below:

- We propose an interpolation scheme that captures the effects of strong scaling an application under different CPU and memory power distributions with minimal profile information.
- We present experimental results showing speedups of up to 2.2X using an overprovisioned system compared to the case where CPU and memory powers are not capped.
- We show the optimized CPU and memory power distributions for different applications and examine the factors that influence them.
- We analyze the effect and importance of base power on achievable speedup for an overprovisioned system.

The rest of the paper is organized as follows. Section **??** describes related work. In Section **??**, we outline our interpolation scheme. Section **??** details our experimental setup. Section **??** presents a case study that demonstrates the working details of our scheme. In Section **??** we present our experimental results.

II. Related Work

To the best of our knowledge, our work is the first study, that estimates and analyzes the optimized *distribution* of power amongst CPU and memory subsystems, in the context of an overprovisioned system under a strict power budget. Rountree et al [**?**] have studied the variation in application performance under varying power bounds using RAPL. In continuation of this work, Patki et al [**?**] proposed the idea of overprovisioning the compute nodes in power-constrained high performance computing. Their work relies on selecting the best configuration out of a set of profiled configurations. Because of the sheer number of all possible configurations, it is not practically feasible to profile an application at all possible node counts, CPU power caps and memory power caps. Our work introduces a novel interpolation scheme, that takes into account the effects of strong scaling an application under different CPU and memory power caps and estimates the missing configurations. Our work also differs from [**?**] since we take into account the effect of memory capping that can significantly improve the speedups for most applications. Another novel aspect is that our scheme is based on *total* machine power which includes base power that can significantly alter the observed speedups across applications.

The idea of overprovisioning has been studied and implemented in the architecture community in a similar context [**?**] e.g. Intel's Nehalem has overprovisioned cores. The CPU can either run all of these cores at lower clock frequencies or a few of them at highest clock frequencies due to power and thermal bounds. Additionally, earlier work has mostly focussed on reducing energy consumption under a time bound for HPC applications. Rountree et al [**?**] have used linear programming to reduce energy consumption with negligible execution time penalty. In our earlier work, we have used DVFS to trade execution time for lower cooling and machine energy consumptions [**?**], [**?**].

III. Approach

Power consumption of different applications varies significantly. Moreover, the usefulness of increasing the power budget of an application also varies between applications [**?**]. We formulate our problem statement as follows:

Optimize the numbers of nodes (n), the CPU power level (p_c) and memory power level (p_m) that minimizes execution time (t) of an application under a strict power budget (P), on a high performance computation cluster with p_b as the base power per node.

In this section, we outline our interpolation scheme that estimates execution time using application profile for different scales, CPU power levels and memory power levels. The terminology used in the paper is defined in Table **??**. We denote an operating configuration by $(n \times p_c, p_m)$ where n is the number of nodes and p_c, p_m are the CPU and memory power caps, respectively. To determine the optimized configuration for running an application, we need to profile the application for each configuration $(n \times p_c, p_m)$ where $n \in N$, $p_c \in P_c$, $p_m \in P_m$. This adds up to a total of $|N| \times |P_c| \times |P_m|$ possible configurations, assuming P_c and P_m have integral values only. Such exhaustive profiling of an application is practically infeasible because of the sheer number of possible configurations. For example, in a cluster with only 20 nodes, 71 CPU power levels and 28 memory power levels, we would need to profile the application for $39,760$ possible configurations, which is practically infeasible. Therefore, we break the application performance analysis into two steps: performance measurement by actual profiling (*Step 1*) followed by performance estimation using curve fitting/ interpolation (*Step 2*).

TABLE I. Terminology

Variable	Description
W	Watts
p_b	node base power (W)
p_c	CPU/Package power cap (W)
p_m	memory power cap (W)
P_c	set of allowed CPU caps used
P_m	set of allowed memory caps used
N	set of number of nodes used
$\mathcal{P}_c$	set of CPU power caps used for profiling in *Step 1*
$\mathcal{P}_m$	set of memory power caps used for profiling in *Step 1*
$\mathcal{N}$	set of number of nodes used for profiling in *Step 1*
P	maximum allowed power budget (W)
t	execution time for an application (s)

Step 1: Performance measurement by actual profiling

We start application profiling by running it for a selected set of configurations that span the entire range of available configurations. In other words, we only profile the application for a subset of total possible configurations i.e. $(n \times p_c, p_m)$ where $n \in \mathcal{N}$, $p_c \in \mathcal{P}_c$, $p_m \in \mathcal{P}_m$.

Step 2: Performance prediction by curve fitting or interpolation

In this step, we use curve fitting on the profiled data obtained in *Step 1* to estimate the execution time for *any* possible configuration $(n \times p_c, p_m)$ where $n \in N$, $p_c \in P_c$ and $p_m \in P_m$. Behavior of execution time (t) with n, p_c, and p_m, can be represented in a 4D plot. However, visualizing a 4D plot can be tedious. Hence, we present application profile by plotting t against the total power p in 2D, where p takes p_c, p_m, and n into account, using the following equation:

$$p = n * (p_b + p_c + p_m) \tag{1}$$

Presenting the profile in a 2D plot facilitates its visualization and makes it easier to determine the optimized configuration. To estimate execution time for any configuration we need to find the relationship of execution time to power consumption across the three dimensions i.e. n, p_c, and p_m. Beginning with $|\mathcal{N}| \times |\mathcal{P}_c| \times |\mathcal{P}_m|$ actually profiled configurations in *Step 1*, interpolation is accomplished in the following three steps:

1) Interpolation across memory power: For each pair of (n, p_c) where $n \in \mathcal{N}, p_c \in \mathcal{P}_c$, we fit a curve $\phi_{n,p_c}(x)$ across the profiled values of memory caps i.e. $p_m \in \mathcal{P}_m$, where $x \in P_m$. This process yields $|\mathcal{N}| \times |\mathcal{P}_c|$ such curves. A given curve, ϕ_{n,p_c}, can be used to obtain an estimate of t corresponding to any $p_m \in P_m$ using n nodes capped at CPU power level of p_c. Using all the ϕ_{n,p_c} curves,

we can estimate the execution times for all configurations $(n \times p_c, p_m)$ where $n \in \mathcal{N}, p_c \in \mathcal{P}_c$, and $p_m \in P_m$.

2) Interpolation across node counts: To capture the behavior of strong scaling, we fit a curve $\psi_{p_c,p_m}(x)$ across the profiled values of n i.e. $n \in \mathcal{N}$, where $x \in N$, for each pair of (p_c, p_m) where $p_c \in \mathcal{P}_c$ and $p_m \in P_m$. This process results in $|\mathcal{P}_c| \times |P_m|$ strong scaling curves. A given strong scaling curve, ψ_{p_c,p_m}, can estimate t for any $n \in N$ where each node is operating under CPU and memory power caps of p_c and p_m respectively. These strong scaling curves can be used to obtain values of t for all configurations $(n \times p_c, p_m)$ where $n \in N, p_c \in \mathcal{P}_c$, and $p_m \in P_m$.

3) Interpolation across CPU power: Finally, we interpolate t across CPU power. We fit a curve $\theta_{n,p_m}(x)$ across the profiled values of p_c ($p_c \in \mathcal{P}_c$) where x$\in P_c$, for every pair of (n, p_m) such that $n \in N$ and $p_m \in P_m$. We retrieve $|N| \times |P_m|$ curves for interpolating across p_c. A given curve, θ_{n,p_m}, estimates t for any $p_c \in P_c$ using n nodes operating under a memory power cap of p_m. These θ_{n,p_m} curves can be used to estimate execution times for all possible configurations i.e. $(n \times p_c, p_m)$, where $n \in N$, $p_c \in P_c$ and $p_m \in P_m$.

IV. Experimental setup

Our testbed is a 20-node Dell PowerEdge R620 cluster installed at the Department of Computer Science, University of Illinois at Urbana-Champaign. Each node is an Intel® Xeon® E5-2620 Sandy-bridge server with 6 physical cores @ 2GHz, 2-way SMT with 16GB of DRAM. The package/CPU corresponds to the processor die that also includes the cores, L1,L2 and L3 caches amongst other components. The Intel Sandy Bridge processor family supports on board power measurement and capping through the Running Average Power Limit (RAPL) interface [?]. The Sandy Bridge architecture has four power planes: Package (PKG), Power Plane 0 (PP0), Power Plane 1 (PP1) and DRAM. RAPL is implemented using a series of Machine Specifics Registers (MSRs) which can be accessed to get power readings for each power plane. RAPL supports power capping PKG, PP0 and DRAM power planes by writing into the relevant MSRs. The package power for our testbed can be capped in the range 25W to 95W (71 integer power levels) while the memory power can be capped between 8W to 35W (28 integer power levels). The average base power per node (p_b) for our cluster was 38 watts. The base power was measured using the in-built power meters on the Power Distribution Unit (PDU) that powers our cluster. Three applications used for demonstrating our scheme are:

1) *Lulesh* - It is a shock hydrodynamics application which was defined and implemented by LLNL as one of five challenge problems in the DARPA UHPC program [?]. This application is both computation and memory intensive.
2) *Wave2D* - uses a finite difference scheme over a 2D discretized grid to calculate the pressure resulting from an initial set of perturbations. This application is both computation and memory intensive.
3) *LeanMD* - is a molecular dynamics simulation written in Charm++ [?]. This benchmark simulates atomic interaction based on Lennard-Jones potential. This application is only computationally intensive [?].

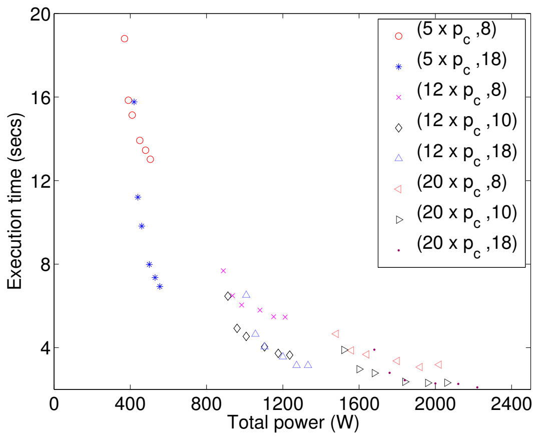


Fig. 1. Average time per step of *L*ulesh for configurations selected in Step 1

V. Case study: Lulesh

In this section, we demonstrate the application of our scheme in estimating the optimized configuration for an iterative application under a strict power budget by considering the *Lulesh* application. . The profiling experiments were conducted on the 20-node cluster outlined in Section **??**. The following CPU and memory power caps were selected for profiling (*Step 1*):

$$\mathcal{P}_c = \{28, 32, 36, 44, 50, 55\}$$
$$\mathcal{P}_m = \{8, 10, 14, 18\}$$

Since determining the optimized number of nodes (n) is part of our scheme, we profile the application for strong scaling as well:

$$\mathcal{N} = \{5, 8, 12, 16, 20\}$$

Figure **??** shows *Lulesh*'s execution profile for some of these configurations. Y-axis corresponds to the average execution time per step, and X-axis shows the *total* power of the system (calculated using Equation **??**). Based on the profile data, we can pick an efficient configuration for a given power budget as follows: We draw a vertical line at the given power budget P and choose the lowest point on or to the left of that vertical line. Following are three examples of finding the optimized configurations for a given power budget (P).

- P = 1200W: the best profiled configuration is $(12 \times 44, 18)$.
- P = 1600W: the best profiled configuration is $(20 \times 32, 10)$. In this case, it is better to use more nodes each capped at relatively lower CPU and memory power levels as compared to the $P = 1200W$ case, in which fewer nodes are run at a higher CPU and memory power levels.
- P = 800W: Since there is no profile data close to the power budget of 800W, we have to proceed leftwards to the $(5 \times 55, 18)$ configuration. This configuration corresponds to a total power consumption of 555W. Hence, the available power is not completely used which makes it an inferior solution.

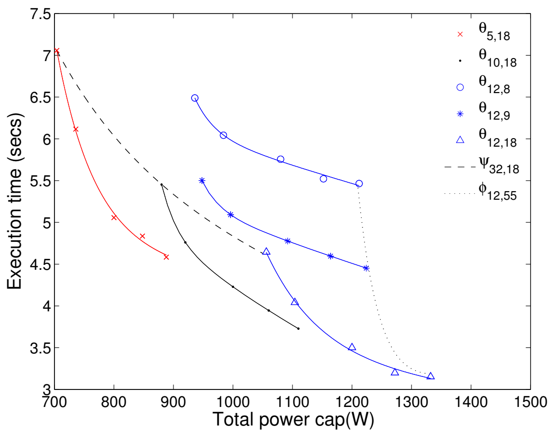


Fig. 2. Average time per step of *Lulesh* after interpolation (Step 2)

We will now see how our interpolation scheme can improve the solution. We need to identify the three sets of functions ϕ_{n,p_m}, ψ_{p_c,p_m}, and θ_{n,p_c} that interpolate across all three dimensions. Figure **??** shows that the behavior of t across all three dimensions is the same. t is more sensitive to each of n, p_c, and p_m at lower values of p as compared to larger values of p. For example, for $n = 12$ and $p_m = 18$, t reduces faster for p_c in range $[28, 36]$W compared to the case when p_c is in the range $[44, 55]$W. Similarly, for $n = 12$ and $p_c = 55$, t reduces faster when p_m is in the range $[8, 10]$W compared to when p_m is in the range $[10, 18]$W. This pattern can be modeled by the use of two exponential terms. We therefore express execution time (t) for each of these curves, $\phi_{n,p_c}, \psi_{p_c,p_m}$ and θ_{n,p_m} by:

$$t(p) = \frac{a}{e^{b*p}} + \frac{c}{e^{d*p}} \qquad (2)$$

where a, b, c, and d are constants and p is total power budget. As mentioned in Section **??**, while interpolating across each of the n, p_c and p_m dimensions, the other two dimensions remain constant. Hence, p (Equation **??**) only captures the change in the dimension being interpolated since the other terms in Equation **??** are constant. We use `Matlab's` curve fitting toolbox that uses linear and non linear regression to determine these constants for each of the curves. Based on the characteristic mentioned above, Equation **??** can be thought of as having two parts: $f_l(p)$ and $f_h(p)$.

$$t(p) = f_l(p) + f_h(p) \qquad (3)$$

For lower values of p, $f_l(p)$ dominates $f_h(p)$, whereas $f_h(p)$ becomes dominating at higher values of p. This is achieved by selecting appropriate constants. At lower values of p, values of t are large and decrease at a faster rate. Hence, the constants a and b in $f_l(p)$ are large. When p is large, t is smaller and decreases slowly with p. This implies smaller values for the constants c and d in $f_h(p)$. For large values of p, a higher value of b also makes $f_l(p)$ negligible.

Figure **??** plots a few of $\phi_{n,p_c}, \psi_{p_c,p_m}$ and θ_{n,p_m} curves for interpolating across the three dimensions i.e. n, p_c, and p_m. To simplify the discussion we omit few profile points from Figure **??**. We remove $p_c = 28W$ from $\mathcal{P}_c$ so that it now is $\mathcal{P}_c = \{32, 36, 44, 50, 55\}$. We now delineate how these curves were obtained by applying *Step 2* described in Section **??**.

- We demonstrate interpolation across memory using the following example. $\phi_{12,55}$ from Figure **??** is obtained by fitting the curve from Equation **??** to configurations $(12 \times 55, p_m)$ for $p_m \in \{8, 10, 14, 18\}$ and evaluating the constants. We can now estimate t for configuration $(12 \times 55, 9)$ using $\phi_{12,55}$ and Equation **??**. This configuration is represented by the rightmost '*' (in blue) in Figure **??**. Similarly, we can fit curves to profile data to obtain the curves ϕ_{12,p_c} for $p_c \in \{32, 36, 44, 50\}$. Using these curves, we can estimate t for configurations $(12 \times p_c, 9)$ for $p_c \in \{32, 36, 44, 50, 55\}$. These configurations are shown by '*' in Figure **??**.
- To estimate the strong scaling performance for different values of n, we do curve fitting for fixed values of p_c and p_m. For example, $\psi_{32,18}$ is obtained by fitting a curve to configurations $(n \times 32, 18)$, for $n \in \{5, 8, 12, 16, 20\}$. We later use this curve to estimate t for $n = 10$. This configuration is represented by the topmost solid black circle in Figure **??**. We obtain the curves $\psi_{p_c,18}$ for $p_c \in \{32, 36, 44, 50\}$ in similar manner and evaluate them at $n = 10$ to estimate t for combinations $(10 \times p_c, 18)$ for $p_c \in \{32, 36, 44, 50\}$ (Figure **??**).
- In the final step, we interpolate data from previous step to get the execution times for all CPU power caps. All the solid lines in Figure **??** correspond to the interpolation across CPU power. For example, $\theta_{5,18}$ is obtained by fitting Equation **??** to configurations $(5 \times p_c, 18)$ for $p_c \in \{32, 36, 44, 50, 55\}$. Finally, we have $|N| \times |P_m|$ curves for θ.

As a result of interpolating across these three dimensions, we now have a *set* of curves, θ_{n,p_m}, that represent *all* possible configurations that could be obtained using exhaustive profiling. To get the optimized configuration for a power budget P, we evaluate all the θ curves for that P and chose the configuration that results in minimum t. If the curve $\theta_{n^0,p_m^0}(P)$ results in the minimum $t = t^0$, the optimized configuration is given by $(n^0 \times \frac{P}{n^0} - p_b - p_m^0, p_m^0)$ after using Equation **??** and solving for p_c.

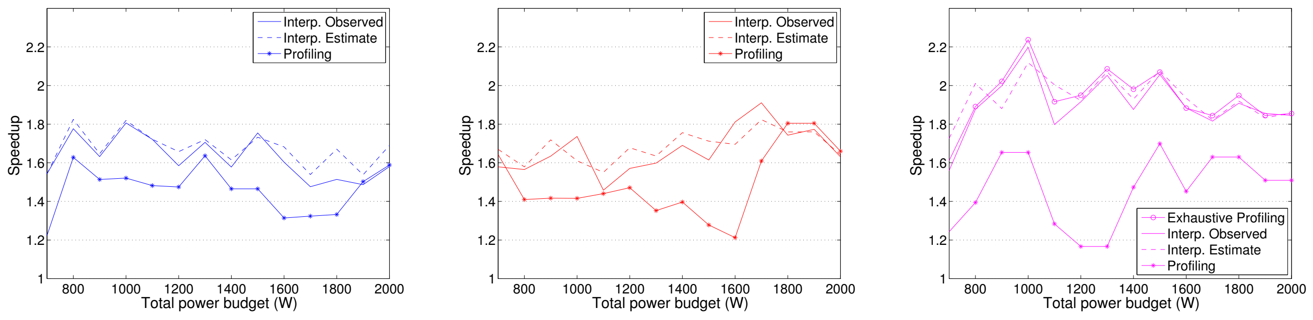
VI. Results

In this section, we use our interpolation model to estimate optimized configurations for different power budgets for the three parallel applications mentioned in Section **??**. Machine vendors specify the thermal design power (TDP) for CPU and Memory subsystems. These numbers represent the maximum power each of these subsystems can draw while operating within the thermal limits. Data centers do not take application characteristics into account and therefore calculate the total power assuming that each node can draw the TDP wattage specified by the manufacturer. We refer to this configuration as the *baseline* configuration . The *baseline* configuration for a power budget of P is given by:

$$(n_b \times TDP_c, TDP_m)$$

$$\text{where } n_b = \left\lfloor \frac{P}{p_b + TDP_c + TDP_m} \right\rfloor,$$

where TDP_c and TDP_m represent the CPU and memory TDP values respectively. For our testbed cluster, TDP_c = 95W whereas $TDP_m = 35$W. TDP of a node for our cluster totals



(a) Lulesh (b) Wave2D (c) LeanMD

Fig. 3. Speedups obtained using CPU and memory power capping in an over-provisioned system

to 168W after adding the base power of 38W. Hence, for the baseline case we employ the maximum number of nodes, without power capping, accounting for a maximum possible power draw of up to 168W per node i.e. $n_b = \frac{P}{168}$. We compare the benefits resulting from power capping CPU and memory subsystems using our scheme against the baseline case for different power budgets. We use speedup over the baseline case as the metric for comparison. Speedup is defined as the ratio of the execution time for the baseline case and the execution time resulting from the optimized configuration estimated by our scheme. We perform real experiments to corroborate the estimates made using our scheme. In particular, we present results to gauge the effectiveness of our approach to meet the following criterion:

- Speedup achieved: Comparing the best configurations from profiled data (Step 1) to the best configurations estimated using our scheme (Step 2).
- Quality of solution: Comparing model estimates to actual experimental results
- Cost of estimating the optimized configuration: Amount of profiling required to make accurate predictions.

A. *Benefits of using our interpolation scheme*

In Figure **??**, we present the speedups achieved using power capping in an over-provisioned system for different power budgets. The 'Profiling' curve plots the actual speedups that are obtained by selecting the optimized configuration from the profiled data from Step 1 (without interpolation). The 'Interp. Estimate' plots the estimated speedups obtained from the interpolated curves from Step 2. We do actual experiments for the optimal configurations predicted in Step 2 and plot the observed speedups in the 'Interp. Observed' curve. We can see from Figure **??** that the observed speedups ('Interp. Observed') match closely to the estimated speedups ('Interp. Estimate'). The difference in the estimated and observed speedups can be mainly attributed to system/cluster noise and to the estimation accuracy of our interpolation scheme. Speedups of *Lulesh*, *Wave2D*, *LeanMD* fall in the range [1.55,1.80], [1.45,1.9], [1.57,2.2] respectively. Although each application ends up in a different speedup range, we get a minimum speedup of at least 1.45X for any power budget. Speedup that an application can achieve is attributed to two factors:

- The difference between the CPU/memory TDP and the *actual* (measured) power consumed by the CPU/memory.
- The sensitivity of execution time to the CPU/memory subsystems power consumption.

Performance can be improved by exploiting the first attribute through a single profiling run. We can profile the application and determine the maximum CPU and memory power consumed by the machine during the execution. However, speeding up the application by exploiting the second factor is only possible if that relationship is known. Figure **??** also compares the speedups for optimized configurations estimated by our model and profiling data. Although using only profiling data can speedup an application, the configurations estimated by our scheme are much superior in terms of speedup. The Observed speedups resulting from our scheme for *LeanMD* are generally 0.40X greater than the configurations estimated by simple profiling (Step 1).

Speedups from just profiling (Step 1) can improve by doing more exhaustive profiling which requires considerable machine time. We mentioned in Section **??** that $|N| \times |P|_c \times |P_m|$ runs are required to profile an application exhaustively. Considering the permissible ranges of p_c, p_m, and n for our testbed, we need to run each application for 39760 configurations, which can be practically infeasible. However, for *leanMD* we did exhaustive profiling since memory power is always less than 8W. It implies that we only need to profile it for different values of n and p_c. The speedups from this exhaustive profiling for *LeanMD* are shown in Figure **??** in the curve labeled 'Exhaustive Profiling'. These speedups are very close to the speedups estimated by our interpolation scheme. This indicates the high accuracy of our scheme in predicting the *optimized* configurations.

B. *Profiling requirements for interpolation*

To analyze the robustness of our scheme in estimating optimized configurations, we used different amounts of profile data as an input to our interpolation scheme. Since there are four unknowns ($a, b, c,$ and d) in Equation **??**, it requires at least 4 data points to fit across each dimension i.e. n, p_c and p_m. Hence, we need at least 64 configurations (data points) for interpolation. We used our scheme to estimate optimized configurations for three different sets of profile data for *Lulesh*. Each profile data set had different number of profiled configurations i.e. 112, 180 and 320 configurations. We used each of these profile data sets as input to our interpolation scheme

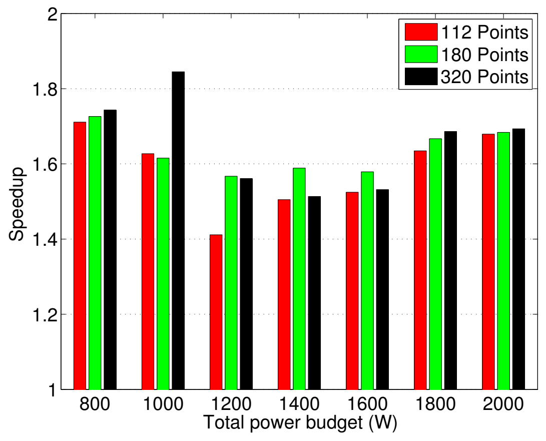


Fig. 4. Observed speedups using different number of profile configurations (points) as input to our interpolation scheme

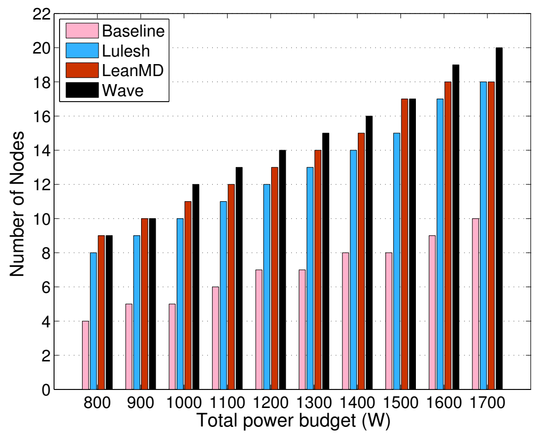


Fig. 6. Optimal number of nodes under different total power budgets

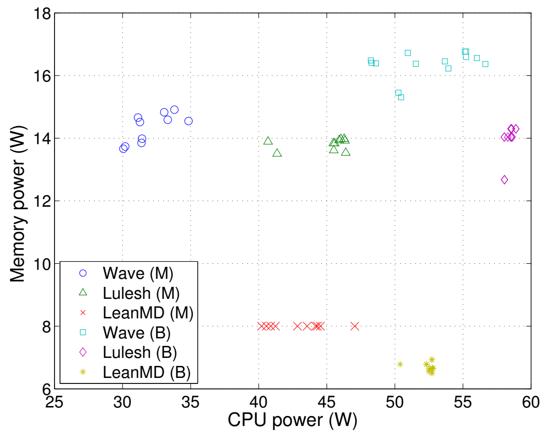


Fig. 5. Optimized CPU and memory power caps under different power budgets compared to the max CPU and memory power drawn in the baseline experiments

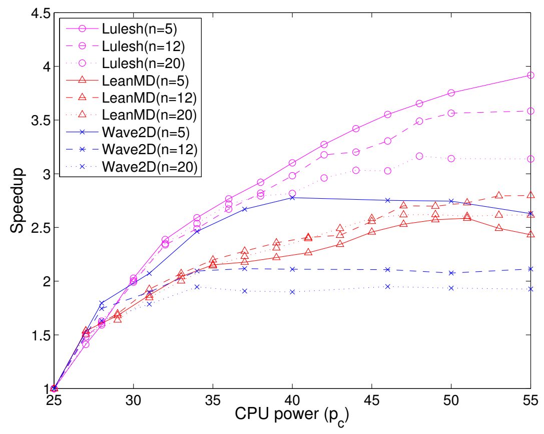


Fig. 7. Speedups over the case of $p_c = 25W$ for different number of nodes (n)

and evaluated the resulting speedups. Figure **??** shows the speedups achieved for various power budgets. These speedups are calculated by performing actual experiments corresponding to the optimized configuration for each case. Although the speedups resulting from optimized configurations generally improve as we increase the profile data points, we are able to achieve reasonable speedups with even 112 configurations.

C. Optimized number of nodes, CPU and memory power distribution

We present the optimized p_c and p_m values resulting from our scheme for different power budgets in Figure **??**. We also plot the *actual (measured)* maximum values for CPU and memory power consumptions in the baseline experiments for the same power budgets. Figure **??** shows that our scheme allocates higher CPU power (p_c) for *Lulesh* and *LeanMD* as compared to *Wave2D*. These optimized values of p_c resulting from our model lie in the range of [29,35] for *Wave2D*. Whereas, the optimized values for p_c range from [41,46] and [40,47] for *Lulesh* and *LeanMD*, respectively. Extra watts allocated to any of the applications outside the upper limit of its range can instead be used to power another node and strong scale the application in an overprovisioned system. Figure **??** shows the optimized number of nodes for each application along for different power budgets and compare it against the baseline configuration. Note that number of nodes in the baseline configuration are independent of the application. Our scheme caps the CPU and memory power to lower values which enables it to use up to twice as many nodes as the baseline case. *Wave2D* generally requires the lowest combined CPU + memory power followed by *LeanMD* and *Lulesh* (Figure **??**). This is why *Wave2D* generally uses the maximum number of nodes followed by *LeanMD* and *Lulesh*.

D. Analyzing the Optimized configurations

The difference in the maximum measured values of CPU power for the baseline case and the optimized value of p_c from our scheme is about 12W for *LeanMD* and *Lulesh* (Figure **??**). This difference is as much as 20W in the case of *Wave2D*. To understand why the two cases are different, we plot speedups for different values of p_c and n. Speedups in Figure **??** are calculated against the execution time at p_c = 25W. This is done to measure the benefits of increasing the CPU power beyond the minimum CPU power cap allowed. *Lulesh* and *LeanMD* are more sensitive to p_c as compared to *Wave2D*. In fact, execution time for *Wave2D* ceases to improve beyond p_c=35W for any value of n. Our interpolation scheme detects this and keeps the optimized value of p_c in the range $[29, 35]$W. Similarly, the curves for *LeanMD* and *Lulesh* in Figure **??** flatten at about 46W and 48W, respectively. This is why the optimized values of p_c are in the range of $[41, 46]$W and $[40, 47]$W for *LeanMD* and *Lulesh*, respectively. For most of the power budgets, the optimized CPU power cap (p_c) for *Lulesh* lies in the range of $[46, 47]$W (barring the two which are close to 41W). This is because of the high sensitivity of *Lulesh* on p_c. Due to this high sensitivity of execution time(t) on p_c, our scheme allocates the highest p_c value for *L*ulesh as compared to the other two applications. In the other two applications, the scheme decides to allocate relatively more nodes rather than increasing p_c, even though every additional node comes with an overhead - its base power.

The optimized memory power from our scheme and the maximum measured memory power in the baseline experiments are almost the same in *LeanMD* and *Lulesh* (Figure **??**). For *LeanMD*, our model caps memory power at 8W which is the lowest memory power cap supported by the machine vendor. Since execution time is highly sensitive to p_m for *Lulesh*, reducing it results in a significant penalty in execution time. Our model captures this sensitivity and suggests p_m that is close to the maximum memory power drawn in case of baseline experiments (14W from Figure **??**). However, in case of *Wave2D*, capping memory power at values less than the maximum power drawn in the baseline scenario, can gives us higher speedups. Figure **??** shows a difference of 2W (on average) between the optimized values of p_m from our scheme and the max memory power drawn in the baseline experiments.

To further explore the reasons for the 20W/2W difference in *Wave2D* CPU/memory power values between our model and the baseline experiments observed in Figure **??**, we study the behavior of CPU and memory power over the course of execution of an application. Figure **??** plots the measured CPU and memory power for the two configurations: $c_1 = (5 \times 55, 18)$ and $c_2 = (5 \times 34, 14)$. The execution time for these configurations is almost the same (within 1% of each other), despite the significant difference in allocated power. Even though the max CPU power drawn reaches 53W for c_1, its average CPU power is just 2W higher than the average CPU power for c_2. Similarly, the max memory power drawn for c_1 is 16W. Capping memory power to 14W in c_2 does not affect the execution time (Figure **??**). Data centers operators have to account for the peak power drawn while deciding how many nodes to use. Due to the fluctuations in both CPU and memory powers for c_1, max power consumed by a node can reach up to 107W $(53 + 16 + 38)$. However, by using configuration c_2, the max power per node can be limited to 86W without any degradation in performance.

E. Benefits of capping memory power

To evaluate the impact of memory power capping, we compared the observed speedups from power capping both CPU and memory (C&M) with the observed speedups from just capping the CPU power (C). In the latter case, it determines the optimized configurations accounting for the maximum TDP wattage of memory i.e. 35W per node. Figure **??** presents the

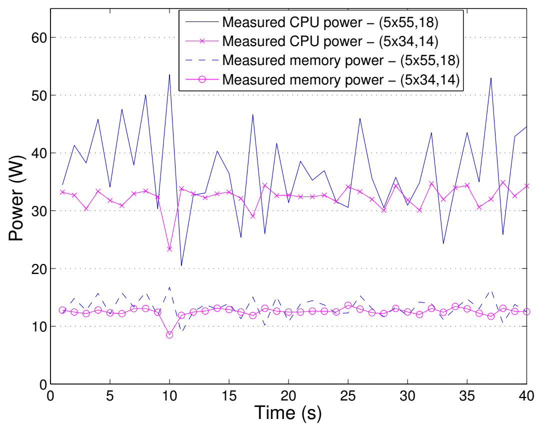


Fig. 8. Measured CPU and memory power for two different configurations with significantly different power allocations but similar execution time

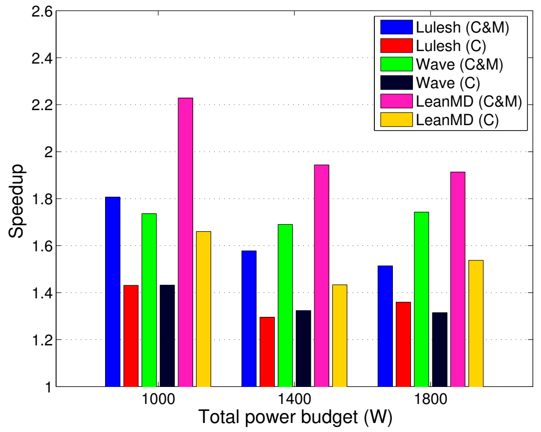


Fig. 9. Speedups for power capping both CPU and memory (C&M) compared to CPU (C) only

speedup results for these two cases under three different power budgets. There is a significant difference between the speedups in the two cases. In case of *LeanMD*, capping memory power increases the speedup from 1.43X to 1.94X for a power budget of 1400W. The ability to cap memory power in addition to CPU power can therefore significantly increase the speedups.

F. Impact of base power on speedups

Base power of nodes play an important role in determining the optimized configuration. It forms an important and essential part of our scheme. Figure **??** shows estimated speedups from our scheme for three different base powers (p_b). These base powers of 10W, 38W, and 60W were measured on the Dell Optiplex 990, Dell PowerEdge R620, and Dell Precision T5500 machines, respectively, using a power meter. As mentioned in Section **??**, the base power of a node in our testbed is 38W. Instead, if the base power was 10W, we can expect the speedups to increase by at least 0.25X. Moreover, the speedups for *Wave2D* and *LeanMD* nearly double if base power is reduced from 60W to 10W. Base power acts as the fixed cost for adding additional nodes in an over-provisioned system. For example, for $P = 800$W, $(15 \times 35, 8)$ and $(7 \times 46, 8)$ are the optimized configurations for *LeanMD* using base

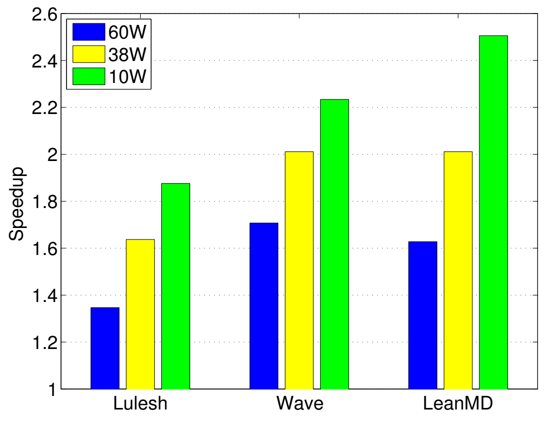


Fig. 10. Estimated speedups for different base powers in case of P=800W

powers of 10W and 60W respectively. For p_b = 10W, our model allocates less power to CPU i.e. p_c=35, and uses 15 nodes. However, increasing p_b to 60W makes it expensive to add more nodes. Hence, for p_b=60W, our model allocates more power to CPU i.e. p_c=46, while using only 7 nodes. This indicates that the optimized configurations shown in Figure **??** would change for different base powers. As the base power increases (decreases), we expect the p_c and p_m from Figure **??** to increase (decrease). We have seen earlier that the optimized configurations depend on the relationship of execution time with p_m and p_c. After looking at Figure **??** we can now associate a correlation between optimum configuration and p_b as well. In general, we can conclude that decreasing the base power increases the speedup.

VII. Conclusions and Future Work

To the best of our knowledge, this paper provides the first experimental study that estimates the *optimized* distribution of CPU and memory power in the context of overprovisioned systems. We first outline our interpolation scheme and describe its working and later use it to estimate optimized configurations for number of nodes and CPU/memory powers. We validate our scheme by comparing its estimated results with actual experimental results. We also establish the quality of solution estimated by our scheme by comparing it to results obtained by exhaustively profiling an application. Our interpolation scheme can speedup an application by up to 2.2X in an overprovisioned system in which the CPU and memory subsystems of individual nodes are power capped as compared to the baseline case in which they are allowed to run at maximum speed (power), under the same global power budget. In this study, we also analyze the effect of base power of a node on the achievable speedups. Our results show that lower base powers can enable overprovisioned systems to achieve higher speedups.

Two compute nodes operating under the same power cap might end up working at different *speeds* due to different thermal conditions. In future, we plan to address this *heterogeneity* by using dynamic load balancing capabilities of of a system that supports dynamic object migration [**?**]. We also plan to investigate the possibility of incorporating thermal constraints along with a strict power constraint. In this paper, we treat an application as an atomic unit. Earlier work shows that different sections of an application can have different sensitivities to CPU power [**?**]. We plan to exploit this behavior of an application by allowing more CPU power to sections of an application that are most sensitive to it, while staying under the total power budget.

VIII. Acknowledgements

This research was supported in part by the US Department of Energy under grant DOE DE-SC0001845 and National Science Foundation under grant NSF ITR-HECURA-0833188. We are thankful to Prof. Tarek Abdelzaher for letting us use the testbed for experimentation under grant NSF CNS 09- 58314.