Comparison of Energy-Constrained Resource Allocation Heuristics under Different Task Management Environments

Bhavesh Khemka*, Ryan Friese*, Sudeep Pasricha*†, Anthony A. Maciejewski*, Howard Jay Siegel*†,
Gregory A. Koenig‡, Sarah Powers‡, Marcia Hilton§, Rajendra Rambharos§, Mike Wright§, and Steve Poole§
*Department of Electrical and Computer Engineering,
†Department of Computer Science,
Colorado State University,
Fort Collins, CO 80523, USA
‡Oak Ridge National Laboratory,
Oak Ridge, TN 37831, USA
§Department of Defense,
Washington, DC 20001, USA

Email: {Bhavesh.Khemka, Ryan.Friese, Sudeep, AAM, HJ}@colostate.edu, {Koenig, PowersSS}@ornl.gov,
mmskizig@verizon.net, Jendra.Rambharos@gmail.com, Michael.Wright4@comcast.net, SWPoole@gmail.com

***Abstract*—There is a growing need for energy-efficiency in high performance computing, especially with systems approaching exascale levels. The Extreme Scale Systems Center at Oak Ridge National Laboratory faces a need for resource management techniques that maximize the performance of the system while satisfying an energy budget. The performance of the system is measured as the total "utility" earned from completing tasks. Utility is represented as a time-varying importance of a task. We perform an in-depth examination into the energy-constrained utility maximization problem by comparing the performance of resource management techniques in two different task management environments: queued and polled. In one environment, tasks are queued for execution on the different machines and certain tasks are not allowed to be re-scheduled. In the other environment, machines are polled at regular intervals and each idle machine is only assigned one task. Multiple First Come First Served heuristics are designed and compared against other heuristics. We design a new adaptive energy filter that can be used with any of the heuristics to bring energy awareness to them. This filtering technique can be readily deployed in any environment without the need of any off-line parameter tuning experiments. The filtering operation allows the heuristics to better regulate their energy expenditure in the energy constrained environment. The polled task management environment and our novel filtering technique give significant performance improvements for the heuristics while meeting the energy budget requirement.**

***Index Terms*—resource allocation; adaptive energy filtering; heterogeneous computing; energy-aware resource management heuristics; system utility**

Corresponding author: Bhavesh Khemka

I. Introduction

The need to solve applications of higher complexity with greater accuracy combined with the need for faster execution from high-performance computing (*HPC*) systems is resulting in higher energy consumption and costs to operate these systems. A recent National Resources Defense Council report showed that data centers in the U.S. consumed an estimated 91 billion kWh in 2013 (that is double the amount of electricity consumed by all of the households in New York City) and are on track to reach 140 billion kWh by 2020 [1]. Some data centers are now unable to increase their computing performance due to physical limitations on the availability of energy. For example, in 2010, Morgan Stanley, a global financial services firm based in New York, was physically unable to draw the energy needed to run a data center in Manhattan [2]. Many HPC systems are now being forced to execute with constraints on the amount of energy they can consume. The issue of increased energy consumption is estimated to significantly worsen as we approach exascale systems. As a result, there is a growing concern regarding energy needed to operate these systems (e.g., [3], [4]) and it is becoming increasingly important for system administrators to adopt energy-efficient workload execution policies.

This research builds on prior work that developed energy-aware resource management techniques with the goal of maximizing the performance of a workload executing on an energy-constrained heterogeneous HPC system [5]. In this new work, we analyze the performance of the resource management techniques in *task management environments* that differ in their policies of which tasks and machines can be considered by the scheduling techniques. We study and contrast a queued and a polled environment. Also, we enhance an energy filter technique (introduced in [5]) by removing the need to determine its parameters empirically and making it adaptive by

using current system information. Such a filtering technique can be deployed directly into any environment without the need to be tuned off-line. The goal of the filtering technique is to ignore allocation choices that use more energy than an estimated fair-share. This improves the distribution of the budgeted energy across the constrained time period.

We model a compute facility and workload of interest to the Extreme Scale Systems Center (*ESSC*) at Oak Ridge National Laboratory (*ORNL*). The ESSC is a joint venture between the United States Department of Defense (*DoD*) and Department of Energy (*DOE*) to perform research and deliver tools, software, and technologies that can be integrated, deployed, and used in both DoD and DOE environments. This system incorporates heterogeneous compute resources that utilize a mix of different machines to execute workloads with diverse computational requirements. In such an environment, each task typically has different execution time and energy consumption characteristics when executed on different machines. We model our machines to have different ACPI performance states (P-states) in which tasks can execute [6].

Each task in the system has a monotonically-decreasing utility function associated with it that represents the task's *utility* (or value) as a function of the task's completion time. The system performance is measured in terms of *total utility earned*, which is the sum of utility earned by all completed tasks [7]. The goal for resource management techniques in this environment is to maximize the amount of utility earned during a period of time that has a constraint on the amount of energy that can be consumed. To keep our simulations tractable, we consider the time period of a day, but one could use any length of time (e.g., six hours, one month, one year). We compare and analyze the performance of our heuristics with different First Come First Served (*FCFS*) heuristics in different task management environments.

In summary, we make the following contributions: (a) the design of a novel adaptive energy filtering mechanism that can be readily deployed into any environment, (b) a comparative analysis of the advantages and disadvantages of a polled task management environment that can be used in HPC environments, and (c) a comparison of multiple FCFS heuristics that are typically used in real schedulers with smarter heuristics that can improve system performance.

The remainder of this paper is organized as follows. The next section discusses our system model and the problem we address. Section III describes the task management environments and our resource management techniques. Our simulation setup is detailed in Section IV. Section V discusses and analyzes our experimental results. We provide an overview of related work in Section VI. We finish with our conclusions and plans for future work in Section VII.

II. Problem Description

A. System Model

In this study, the system model is similar to the model described in [5]. We model a dynamic system where tasks arrive throughout the day and a resource manager maps the tasks to machines for execution. We consider an *oversubscribed* environment, i.e., the incoming workload exceeds the capacity of the computing system. The workload and computing system we model are based on the interests of the ESSC. Each task in the system has an associated utility function (introduced in [7]). The utility function of a task is a monotonically-decreasing function that represents the "value" of completing that task at different times. Figure 1 shows an example utility function for a task and highlights the utility that will be earned when the task is completed at two different times. The utility function of a task is assumed to be set by the user in collaboration with the system owner. The utility functions are described by three parameters: priority, urgency, and utility class [7]. *Priority* controls the overall importance of the task by setting the task's maximum (i.e., starting) utility value. *Urgency* sets the overall rate of decay for the utility function. The *utility class* allows the shape of the utility function to be modified by partitioning it into intervals and specifying shape modifiers for each of the intervals.

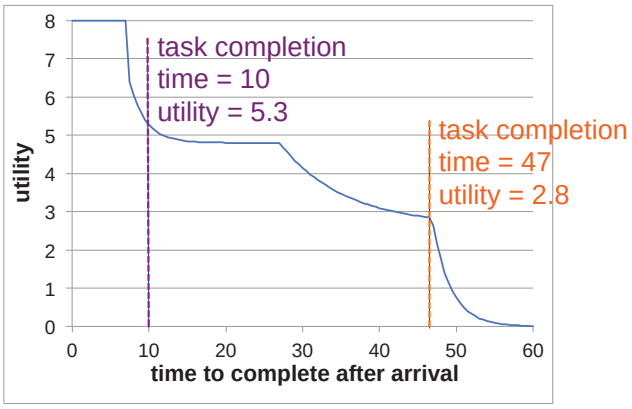

Fig. 1. A sample utility function showing the utility earned at two different completion times.

Our computing system environment consists of heterogeneous machines, where each machine belongs to a specific *machine type* (rather than a single large monolithic system, such as Titan [8]). Machines belonging to different machine types may differ in their microarchitectures, memory modules, and other system components. We model the machines to contain CPUs with dynamic voltage and frequency scaling (*DVFS*) enabled to utilize different ACPI performance states (*P-states*) that offer a trade-off between execution time and power consumption. We group tasks with similar execution time and power characteristics into *task types*. Tasks belonging to different task types may differ in characteristics such as computational intensity, memory intensity, I/O intensity, and memory access pattern. The application (task) developer specifies which task type the application falls into. The type of a task is not related to the utility function of the task. Because the system is heterogeneous, machine type A may be faster (or more energy-efficient) than machine type B for certain task types but slower (or less energy-efficient) for others. We model

general-purpose machine types and *special-purpose* machine types in our heterogeneous system [9]. The special-purpose machine types execute certain special-purpose task types much faster than the general-purpose machine types, although they may be incapable of executing the other task types.

We assume that for a task of type i on a machine of type j running in P-state k, we are given the *E*stimated *T*ime to *C*ompute ($ETC(i,j,k)$) and the *A*verage *P*ower *C*onsumption ($APC(i,j,k)$). It is common in the resource management literature to assume the availability of this information based on historical data or experiments [10]–[16]. The APC incorporates both the static power (not affected by the P-state of the task) and the dynamic power (different for different P-states). We can compute the *E*stimated *E*nergy *C*onsumption ($EEC(i,j,k)$) by taking the product of execution time and average power consumption, i.e., $EEC(i,j,k) = ETC(i,j,k) \times APC(i,j,k)$. ETC and APC values for the ESSC environment are not available to researchers. Therefore, for the simulation study conducted in this paper, we synthetically create ETC and APC matrices based on recommendations provided by ESSC and based on general trends of the workloads.

Tasks are assumed to be independent (they do not require inter-task communication) and can execute concurrently (each on a single machine, possibly with parallel threads). This is typical of many environments, such as [17]. We do not allow the preemption of tasks, i.e., once a task starts execution, it must execute until completion.

B. Problem Statement

With tasks dynamically arriving, the scheduler does not know the arrival time, type, or utility function of the next task. *The goal of the scheduler is to maximize the total utility that can be earned from completing tasks during a given period of time while satisfying an energy constraint (E) for that time period.* We use the duration of a day to keep the simulation time tractable. Instead of one day we could base our constraint on any interval of time (e.g., two hours, six months, a year). For ESSC, constraints on power (energy per time) are not a concern. As the system modeled is oversubscribed, the machines are never turned off, and therefore, the fraction of static versus dynamic power is not relevant.

III. Resource Management

A. Overview

Heuristics are commonly used to solve task to machine scheduling problems that have been shown to be NP-hard [18]. A *mapping event* occurs any time a scheduling decision has to be made. We use *batch-mode heuristics* that trigger mapping events after fixed time durations (one minute in our environment) after the previous mapping event completes [7], [19].

During a mapping event, three decision processes are executed. The first operation drops tasks that have low potential utility at the current time to allow the system to better tolerate high oversubcription scenarios. The second operation is an energy filtering technique. We adapt the technique from our work in [5] to control the energy expenditure by preventing tasks from using more than their "fair-share" of energy. We enhance the energy filtering technique to use more system information and enable it to automatically adjust its level of energy filtering without the need for any parameter tuning. The final operation during a mapping event does the actual mapping of tasks to machines in certain P-states using some heuristic approach.

We study two different task management environments: queued and polled. In the *queued* environment, each of the machines has a queue of tasks that it will execute in the queue order. The task that is next-in-line for execution on a machine is referred to as the *pending task*. All other tasks that are queued for the machines are said to be in the *virtual queues* of the scheduler. Figure 2a shows the state of a small example system prior to a mapping event with four machines and executing tasks, tasks in the pending slots, the virtual queues of the scheduler, and the tasks that have arrived since the last mapping event. At a mapping event in the queued environment, the batch-mode heuristics make scheduling decisions for a set of tasks comprising those that have arrived since the last mapping event and the ones that are currently in the virtual queues. This set of tasks is called the *mappable tasks set*. The batch-mode heuristics are not allowed to remap the pending tasks so that the machines do not idle if the currently executing tasks complete while the heuristic is executing. As we do not allow preemption, the currently executing tasks cannot be interrupted and therefore are not available for mapping. We refer to the pending and the currently executing tasks as the *unmovable tasks*. As there are queues for the machines, the heuristics consider all machines as *available* choices when performing mapping decisions. Figure 2b shows an example state of the machines immediately after the mapping event is performed.

In the *polled* environment, individual machines do not have queues. Rather, at each mapping event, the machines are polled to check if they are currently idle or if they are executing a task. Only the machines that are idle are considered to be "available" for scheduling during the mapping event. The "mappable tasks set" in this environment comprises tasks that were either unmapped in the previous mapping event or newly arrived tasks since the last mapping event. In the polled environment, the only "unmovable tasks" are the tasks that are currently executing. Figure 3a shows an example state of a four-machine system in a polled environment prior to a mapping event, and Figure 3b shows a possible state of the system immediately after the mapping event.

B. Dropping Low Utility Earning Tasks

We use a technique to drop tasks with low potential utility at the current time (introduced in our previous work [7]). *Dropping* a task means that it will never be mapped to a machine. Due to the oversubscribed environment, if a resource allocation heuristic tried to have all tasks execute, most of the task completion times would be so long that the utility of most tasks would decay significantly and be very small. This

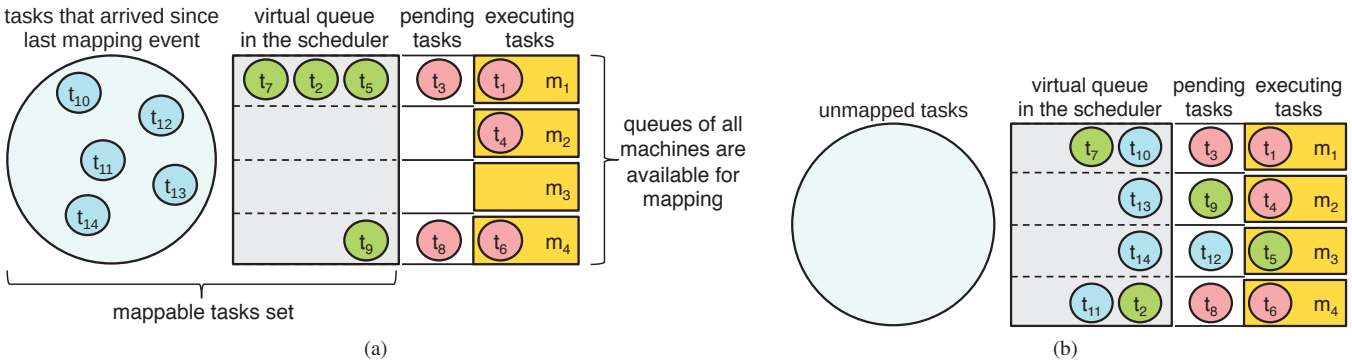

Fig. 2. Example state of a four-machine system in the "queued" environment (a) before and (b) immediately after a mapping event.

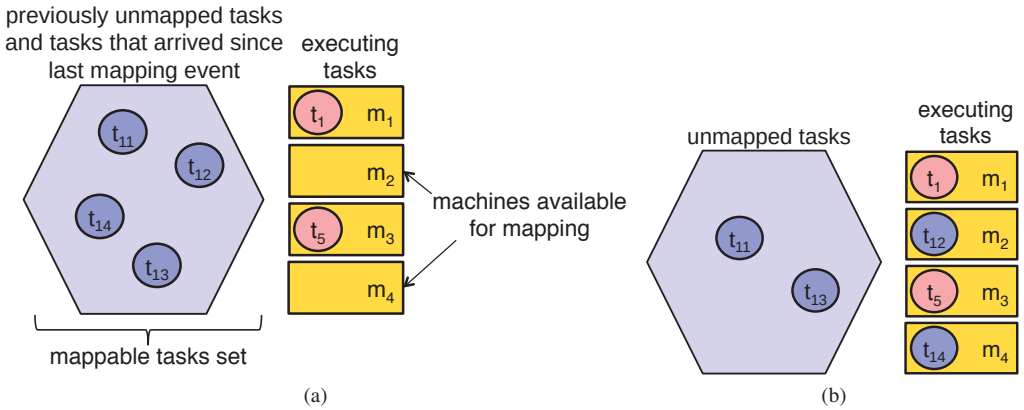

Fig. 3. Example state of a four-machine system in the "polled" environment (a) before and (b) immediately after a mapping event.

would negatively impact users as well as the overall system performance. Given that the performance measure is the total utility achieved by summing the utilities of the completed tasks, dropping tasks leads to higher system performance, as well as more users that are satisfied.

The dropping operation determines the maximum possible utility that each mappable task could earn on any available machine assuming it can start as soon as possible, i.e., immediately after the unmovable tasks. If this utility is less than a *dropping threshold* (determined empirically), we drop this task from the set of mappable tasks. If the utility earned is not less than the threshold, the task remains in the mappable tasks set and is considered in the subsequent allocation decisions of the mapping event.

As our environment is oversubscribed, the number of tasks in the mappable tasks set increases quickly. With more tasks in the mappable set, the heuristics may take longer to perform their mapping decisions. This can delay the trigger of subsequent mapping events and result in poor performance because any newly arrived high utility tasks may not get serviced in a timely manner. Therefore, by dropping tasks with low potential utility, we reduce the size of the mappable tasks set and enable the heuristics to complete their execution quicker and as a result trigger subsequent mapping events sooner. This allows the heuristics to promptly service any newly arriving high utility-earning tasks.

C. Energy Filtering

The goal of our energy filter technique is to remove potential allocation choices (task/machine/P-state combinations) from a heuristic's consideration if the allocation choice consumes more energy than an estimated fair-share energy budget (e_{bud}). The value of the e_{bud} needs to adapt based on the energy remaining in the day and the time remaining in the day. Therefore, the value of the e_{bud} is recomputed at the start of every mapping event.

We denote e_{cons} as the total energy that has been consumed by the system in the current day, and e_{unmov} as the energy that is guaranteed to be consumed, i.e., by unmovable tasks. The total energy that can be scheduled by heuristics (without violating the day's energy constraint) is denoted by e_{rem}. It is computed as $e_{rem} = E - (e_{cons} + e_{unmov})$.

To estimate e_{bud}, the filter also needs to compute the time remaining in the day within which the above energy can be consumed. The *availability time* of a machine is set to either the completion time of the last unmovable task on the machine or the current time, whichever is later. We compute the total time remaining for computations (τ_{rem}) by summing across machines the difference between the end time of the day and the availability time of the machine.

The average of the execution time values and energy values of all task types, machine types, and P-states is represented as $\bar{\tau}$ and $\bar{e}$, respectively. The energy filtering technique needs

to estimate the total number of tasks that it can execute until the end of the day on average. Based on the time remaining, the estimated number of tasks that can complete on average was calculated as $\tau_{rem}/\bar{\tau}$ [5]. Similarly, based on the energy remaining, we now estimate the number of tasks that can complete on average as $e_{rem}/\bar{e}$, and use the minimum of these two ratios as the number of tasks that the system can complete on average (n):

$$n = min\left(\frac{\tau_{rem}}{\bar{\tau}}, \frac{e_{rem}}{\bar{e}}\right). \quad (1)$$

To control the strictness of the filtering technique, we use a multiplier (λ). e_{bud} is computed using:

$$e_{bud} = \lambda \times \frac{e_{rem}}{n}. \quad (2)$$

When n is determined by the energy remaining in the system in Equation 1 ($e_{rem}/\bar{e}$), the second term of the product in Equation 2 appropriately reduces to using the average energy consumption of a task ($\bar{e}$) to determine e_{bud}.

Instead of using a fixed value for λ that needs to be empirically determined by running multiple parameter tuning experiments [5], we enable it to adapt based on the current rate of energy consumption and the target energy consumption rate. We denote the total compute time available in the system at the start of the day as T and its value is calculated by simply multiplying the number of machines by the total time in a day. The adaptive value for λ (which is recomputed at the start of every mapping event) is calculated as:

$$\lambda = \frac{E/T}{\left(\frac{e_{cons} + e_{unmov}}{T - \tau_{rem}}\right)}. \quad (3)$$

The goal of this adaptive parameter is to distribute the energy usage throughout the day. The numerator in Equation 3 is the target energy consumption rate and the denominator is the current average rate of energy consumption. If the current average rate is lower than the target rate, then that leads to a larger value for λ, which allows more energy to be consumed by allocation choices. If the current average rate is higher than the target, the resulting smaller value of λ will only let low energy-consuming allocation choices pass through the filter. In this way, this adaptive technique automatically adjusts the level of filtering and can be deployed readily into any environment without the need for extensive off-line parameter tuning experiments.

D. Heuristics

We use the dropping operation with all the heuristics. We analyze the performance of the heuristics with and without the energy filtering technique. The heuristics are given the tasks and the task/machine/P-state choices that passed the dropping operation and the energy filtering technique (if used) to finally make assignments of the mappable tasks to the available machines. The *ready time* of a machine is the time by which it completes execution of the last task that is queued on it. In the polled environment, the ready time of the available machines is simply the current time. All of the heuristics progress iteratively, and in each iteration they make assignment for a mappable task to an available machine. The heuristics stop executing if either the set of mappable tasks is empty, or if there are no more available machines. In this work, we compare and analyze the performance of twelve heuristics. These heuristics assign tasks during mapping events while there still remains energy in the day. All heuristics ensure that the allocation they plan to make is a *valid assignment*, i.e., not assigning a task that cannot run on a particular special-purpose machine.

As most real-world schedulers assign tasks in an order based on their arrival time, we design and study the performance of a First Come First Served (*FCFS*) heuristic. The FCFS heuristic maintains a list of mappable tasks in an ascending order of arrival time. It then iteratively works through the list, each time making an assignment for the first task in the list. For the task being considered in each iteration, the heuristic assigns it to the available machine that has the earliest ready time that is both a valid assignment, and that has passed the energy filter in the fastest P-state (i.e., P-state 0). In the polled environment, the heuristic picks a machine that is immediately available. The assigned task is removed from the sorted list and the next task is considered. We call this the ***FCFS P-state 0*** heuristic. We implemented another version of this heuristic that examines if the slower P-states pass through the energy filter in case the fastest P-state does not. We call this the ***FCFS All P-states*** heuristic.

The system examined in this study is oversubscribed, and the utility of tasks may start to decay once they arrive. Therefore, we consider an alternative to the FCFS heuristic that gives higher preference to the latest arrived task. This is the Last Come First Served (*LCFS*) heuristic. The LCFS heuristic is similar to the FCFS heuristic except that it maintains a list of mappable tasks in a descending order of their arrival times. We call the version that only permits the fastest P-state as the ***LCFS P-state 0*** heuristic and the version that allows any P-state to be chosen as the ***LCFS All P-states*** heuristic.

To account for the different importance levels of the tasks, we design prioritized versions of the FCFS and the LCFS heuristics. We envision that a version of these heuristics are implemented in real-world schedulers where the submitted jobs have different priority levels. The ***Prioritized-FCFS P-state 0*** heuristic first groups the mappable tasks based on their priority level (i.e., value of their initial maximum utility). It maintains a list of the tasks within each group in an ascending order of arrival time. The heuristic then considers the highest priority level group that contains any tasks. Considering those tasks in order, it makes assignment for each task to the earliest available machine that is a valid assignment and that has passed through the energy filter in P-state 0. After considering all the tasks in this group, it then considers the next highest priority level group that contains any tasks and continually repeats this process. We designed another version of this heuristic that allows the other P-states to be considered as well. We call this the ***Prioritized-FCFS All P-states*** heuristic.

The prioritized versions of the LCFS heuristics are called the ***Prioritized-LCFS P-state 0*** and ***Prioritized-LCFS All P-states*** heuristics. These heuristics are similar to the Prioritized-FCFS heuristics with the difference that they maintain a list of tasks within each group in a descending order of their arrival times.

The ***Max Utility*** heuristic (designed based on the Min-Min technique [19]–[21]) gives preferences to mapping choices that earn the highest utility. The heuristic computes the utility that will be earned by each of the mappable tasks on each of the available machines in the different P-states. Each mappable task independently finds the machine/P-state choice that is valid, passes the energy filter, and that maximizes the utility earned. Among the different task/machine/P-state choices found, an assignment is made for the choice that earns the highest utility. The assigned task is removed from the set of mappable tasks and the process is repeated. This heuristic examines the utility function to find the utility that will be earned at the task completion time as opposed to the Prioritized heuristics that only look at the starting utility value of the task.

Unlike the Max Utility heuristic that solely maximizes for utility, the Max Utility-per-time (***Max UPT***) heuristic picks the task/machine/P-state choice that maximizes the ratio: "utility earned / execution time." In an oversubscribed environment, it is important to consider how much utility is earned per execution time used.

The Max Utility-per-Energy (***Max UPE***) heuristic focuses on reducing energy along with maximizing utility earned. It picks the allocation choice that maximizes the ratio: "utility earned / energy consumed by allocation." In an energy-constrained environment, this heuristic helps to rank allocation choices by considering both the worth of a task and its energy consumption.

For comparison purposes, we implement a ***Random*** heuristic that randomly assigns a mappable task to an available machine in a random P-state (among the allocation choices that passed through the energy filter and are valid).

IV. Simulation Setup

A. Overview

We use simulations to study our problem because we want to test and analyze the performance of the heuristics in a variety of environmental conditions. We simulate the arrival and mapping of tasks over a duration of 26 hours, with the first two hours used to bring the system up to steady-state operation. We collect our results (e.g., total utility earned, energy consumed) only from the start of the third hour to the end of the 26$^{\text{th}}$ hour (total of 24 hours) to avoid the scenario where the machines start with empty queues. All the simulation experiments were run on the ISTeC Cray System at Colorado State University [17]. Each of the trials represents a new workload of tasks (with different utility functions, task types, and arrival times), and a different computing environment by using new values for the entries in the ETC and APC matrices (but without changing the number of machines). All of the parameters used in our simulations are set to closely match the expectations for future environments of interest to the ESSC.

B. Workload Generation

A utility function for each task in a workload is given, and each task has a maximum utility value (depending on its priority level) that starts at one of 8, 4, 2, or 1. These values are based on the plans of the ESSC, but for other environments, different number of values and different values of maximum utility may be used. A method for generating utility functions can be found in [7]. Each task belongs to one among four urgency levels and 20 utility classes.

In our simulation environment, approximately 32,000 tasks arrive during the duration of a day, and each belongs to one of 100 task types. Out of the 100 task types, 83 are general-purpose and 17 are special-purpose. Each task type has approximately the same number of tasks in it. We generate the arrival patterns to closely match patterns of interest to ESSC [7]. The general-purpose tasks arrive in a sinusoidal pattern and special-purpose tasks follow a bursty arrival pattern.

C. Execution Time and Power Modeling

The compute system that we model has 13 machine types (four special-purpose) consisting of a total of 100 machines. The four special-purpose machine types have 2, 2, 3, and 3 machines in them. The remaining 90 machines are general-purpose and are split into the remaining nine machine types as follows: 5, 5, 5, 10, 10, 10, 10, 15, and 20. The machines of a special-purpose machine type run a subset of special-purpose task types approximately ten times faster on average than the general-purpose machines can run them (as discussed below). The special-purpose machines do not have the ability to run tasks of other task types. In our environment, three to five special-purpose task types are special for each special-purpose machine type.

We assume that heuristics can make use of three P-states in all machines: the highest power P-state (P-state 0), lowest power P-state, and an intermediate P-state. We use techniques from the Coefficient of Variation (*COV*) method [22] to generate the entries of the ETC and APC matrices in the highest power P-state. The mean value of execution time on the general-purpose and the special-purpose machine types is set to ten minutes and one minute, respectively. The mean dynamic power was set to 133 watts. To generate the dynamic power values for the intermediate P-state and the lowest power P-state, we scale the dynamic power to 75% and 50%, respectively, of the highest power P-state. The execution times for these P-states are also generated by scaling the execution time at the highest power P-state by sampling a gamma distribution with a mean value that is approximately $1/\sqrt{(\%\ scaled\ in\ power)}$. For example, the lowest power P-state's execution time will be scaled by a value sampled from a gamma distribution that has a mean approximately equal to $1/\sqrt{0.5}$. The execution time of any task is guaranteed to be the shortest in the highest power P-state, but the most energy-efficient P-state can vary across tasks. Such a model

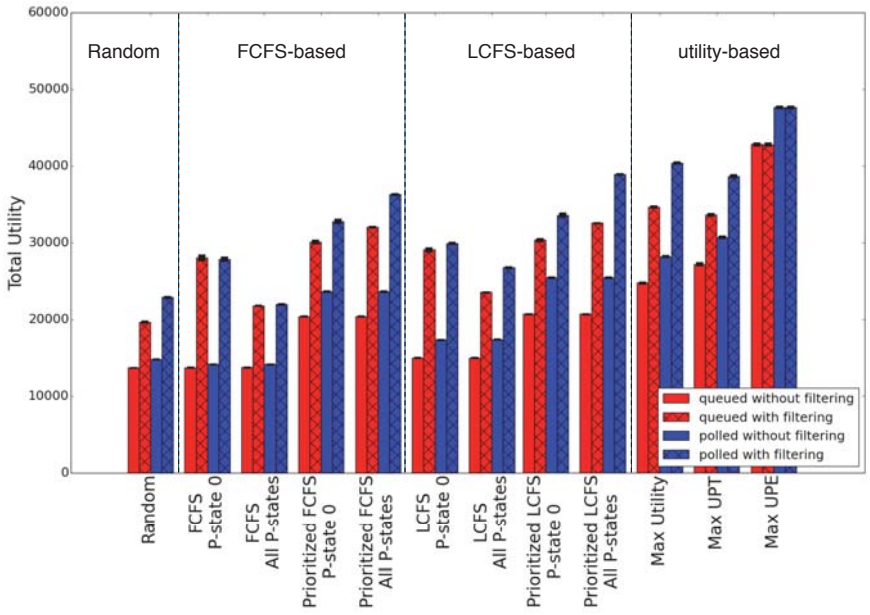

Fig. 4. Total utility earned by the heuristics in the queued and the polled task management environments. For each of those cases, the hatched bars show the performance when the adaptive energy filter technique was used. For almost all the heuristics, the polled environment and the energy filtering technique help to significantly improve performance. The results are averaged over 48 simulation trials and 95% confidence intervals are computed.

approximates reality where the impact on execution time and energy consumption by switching P-states depends on, among other factors, the CPU-intensity/memory-intensity of the task and static power of the system.

D. Obtaining an Energy Constraint

To obtain an energy constraint for use in our simulation studies, we applied a heuristic from our previous work that did not have an energy constraint. In particular, we used Max UPT because it maximized utility in [7], where energy was not a constraint. We used the same workload and environment setup as mentioned in Sections IV-B and IV-C and recorded how much energy was consumed in the day when using Max UPT to make scheduling decisions without any constraint on energy. We determined this energy value for the 48 simulation trials and computed its average. To make our problem challenging, we set the energy constraint for our environment to be 70% of this average value. As a result, for our simulations, we used an energy constraint value of 1.11 GJ per day.

V. Results

All results shown in this section display the average over 48 simulation trials with 95% confidence interval bars. The execution time of the heuristics and other mapping operations is on the order of 10^{-4} seconds per mapping event with the maximum time being ≈ 3 milliseconds. Therefore, in all of our cases, the mapping events were triggered approximately every 60 seconds. All of the heuristics used a dropping threshold of 0.5 units of utility to tolerate the oversubscription, as it gave the best performance without removing tasks of any one priority level completely. The dropping operation helped reduce the execution time of heuristics in the oversubscribed environment. When selecting a dropping threshold, one must consider the level of oversubscription of the environment in addition to the utility values of tasks.

Figure 4 shows the utility earned by the various heuristics in both queued and polled environments as well as with and without the energy filtering technique. The dashed vertical lines separate heuristic types into the following: Random, FCFS-based heuristics, LCFS-based heuristics, and the heuristics that use utility function information. We observe that, in general, the performance of the heuristics on the right is better than those on the left. The Prioritized FCFS heuristic performs better than the FCFS heuristic because it focuses on executing the high priority tasks first, and as a result, earns more utility per task. This is useful in an oversubscribed system as the heuristic is required to pick the better tasks to run. We see a similar trend when comparing Prioritized LCFS with LCFS. The LCFS-based heuristics usually outperform their FCFS counterparts because the LCFS-based heuristics focus on serving more recently arrived tasks. These recently arrived tasks tend to have utility values that have decayed less than tasks that arrived in the system earlier. The best performance is obtained by the heuristics that use utility function information to determine how much utility a task will earn. This is important because even though a task may have the highest priority it may also decay very fast. By the time such a task completes it may earn less utility than a task that may have a lower priority but does not decay much or does so slowly. The Max UPE heuristic significantly outperforms the other heuristics because it proactively reduces energy consumption and uses the saved energy to execute more

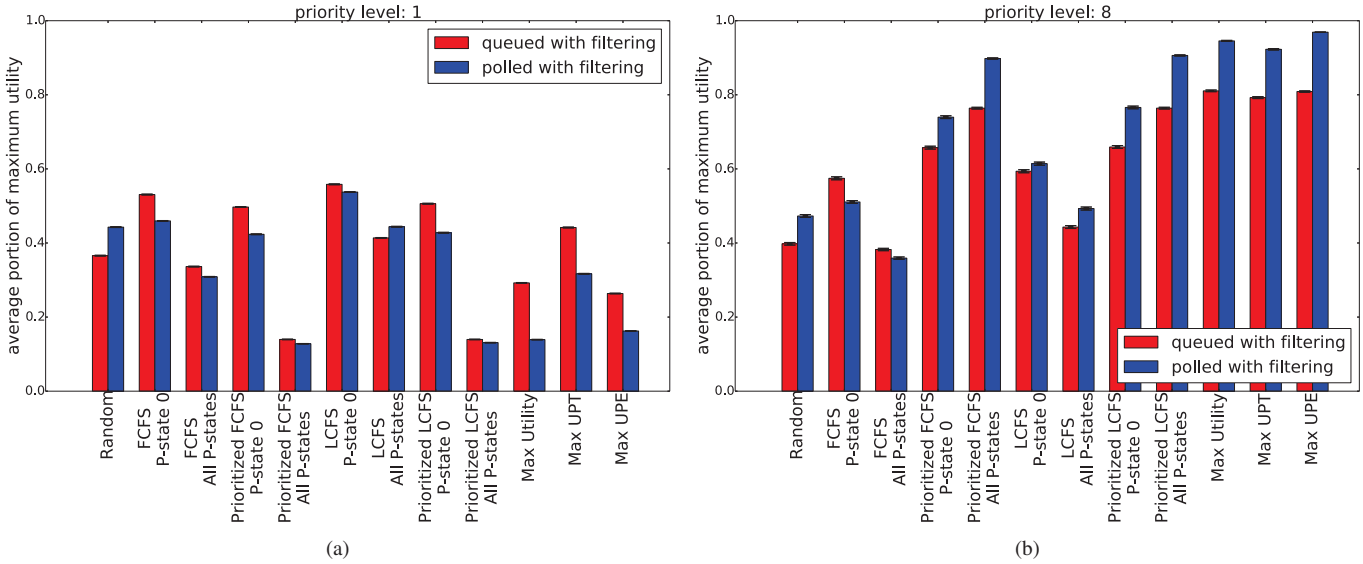

Fig. 5. Portion of maximum utility earned by the different heuristics when using the filtering technique for tasks that belong to (a) priority level 1, and (b) priority level 8.

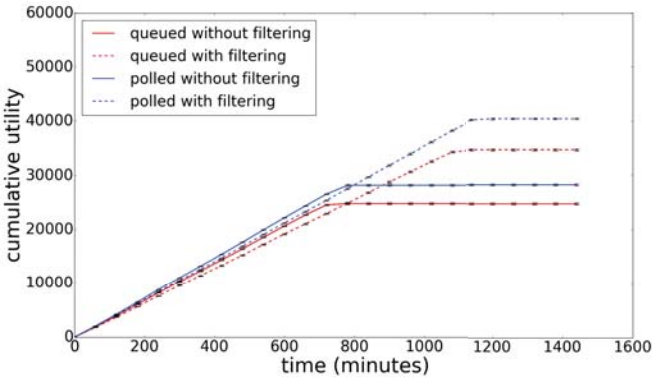

Fig. 6. Cumulative utility earned by the Max Utility heuristic as time progresses highlighting the benefit of the filtering technique to save energy and use it to earn utility in the later parts of the day. The results are averaged over 48 simulation trials and 95% confidence intervals are computed.

tasks and earn more utility.

We observe that the polled environment significantly improves the performance for most of the heuristics, compared to the queued environment. This happens because the polled environment does not lock down a task into the pending slot, as happens in the queued environment, and therefore, is better able to more quickly serve any high utility earning tasks that arrive compared to the queued environment. It is worth noting that the polled environment has more idle time than the queued environment. This is because, in the polled environment, whenever a machine finishes execution of a task, it has to wait (idle) until the trigger of the next mapping event for another task to be assigned to it. Therefore, traditional metrics for performance such as utilization would incorrectly identify the polled environment to be performing worse, but it is those small amounts of idling distributed throughout the day that improves the ability of the system to quickly service any high utility tasks that may arrive. Figures 5a and 5b show on average the portion of maximum utility that was earned from tasks that belonged to priority level 1 and priority level 8, respectively. We see that for tasks that belong to the higher priority level, the polled environment does a better job of earning higher utility as compared to the queued environment. The figures also highlight the effectiveness of the priority-based and utility function-aware heuristics in attempting to earn a larger portion of the maximum utility from tasks that have a higher priority level versus those that have a priority level of 1. The trend with a priority level of 4 is very similar to 8 and the trend with a priority level of 2 is in-between the trends in the priority level 8 and 1 charts. The results shown in these figures are for the cases using the filtering technique but the no filtering cases also show similar overall trends.

It is worth noting that the time between subsequent mapping events versus the average task execution time is an important factor in determining which type of task management environment would be the best. For example, if the heuristic execution times were to be very long, leading to longer times between mapping events, then it is likely that the queued environment will perform better than the polled environment because the polled environment will starve the machines whereas in the queued environment the machines can at least allocate tasks based on the queues.

Figure 4 also shows the significant benefit provided by our novel adaptive energy filtering technique for almost all of the heuristics in either of the task management environments. The filtering technique allows the heuristics to save energy for the later part of the day that can then be used to execute and earn utility from any high utility tasks that arrive at those

times. A trace chart showing the utility earned by the Max Utility heuristic as time progresses (Figure 6) highlights the benefit of using the filtering technique. In our ESSC inspired environment, high-utility tasks arrive throughout the day and therefore it is typically beneficial to have some energy left to spend towards the end of the day. Similar trends are observed for the other heuristics as well and our filtering technique helps all of the heuristics in a similar manner by adapting to their energy consumption rate. When not using the filtering technique, all of the heuristics except Max UPE hit the energy constraint before the end of the day, and therefore, the heuristics (except Max UPE) benefit from using the energy filter. As Max UPE already considers energy consumption, it does not benefit from the energy filter. Figure 6 shows how the polled environment always has a slightly higher slope than the queued environment. This is because the polled environment is consistently better able to serve the high priority tasks that arrive than the queued environment is able to serve.

For the FCFS-based and the LCFS-based heuristics, whether only P-state 0 (fastest P-state) is permitted or if all the other P-states are allowed only matters in the filtering cases, because in the no filtering case, the heuristic always makes assignments in P-state 0. When using filtering with the FCFS and LCFS heuristics, Figure 4 shows that considering only P-state 0 has a larger improvement in performance compared to the performance improvement when considering all P-states. This happens because of two reasons. The first reason is that by considering other P-states in all of the machines, even though more energy is saved, the execution time also increases leading to only a limited increase in total utility. It is worth noting that the first choice that satisfies the filter is chosen as opposed to the best energy choice. The second reason is that by only considering P-state 0, the FCFS and LCFS heuristics get the benefit of searching for another machine that satisfies the energy filter (but still only uses the fastest P-state). This search for a low energy machine also leads to minimizing the execution time in our environment (as energy = power $\times$ time). This results in more utility being earned from task completions. These benefits are not substantial when using the Prioritized FCFS and Prioritized LCFS heuristics, as they first consider the highest priority tasks that are not necessarily the earliest arriving or latest arriving, and the benefit from considering only P-state 0 is less than the benefit of the energy savings provided by considering other P-states, allowing them to earn more utility during the later part of the day.

VI. Related Work

In [7], the concept of utility functions to describe a task's time-varying importance is introduced. Energy is not considered at all in that paper. In this work, we are concerned with maximizing utility while obeying an energy constraint.

Energy-aware scheduling has been extensively studied. In [23], the authors design techniques to schedule a bag-of-tasks to a heterogeneous computing system with the goal of minimizing energy consumption under a throughput constraint. In [24], the authors formulate a bi-objective resource allocation problem to analyze the trade-offs between makespan and energy consumption. Our work differs from these as we maximize utility earned under an energy constraint.

In [25], a set of dynamically arriving tasks with individual deadlines are allocated to machines within a cluster environment with the goal of conserving energy. Specifically, the authors try to optimize the energy consumption while meeting the constraint of completing all tasks by their deadlines. Our environment tries to maximize the total utility earned while operating under an energy constraint. As a result, we design an adaptive energy filtering technique. Additionally, [25] models an undersubscribed system, while our work focuses on highly oversubscribed environments.

The research in [26] attempts to maximize a mathematical model of Quality of Service under an energy constraint by using DVFS to take up slack time in an undersubscribed system, which is very different from our oversubscribed environment.

A dynamic resource allocation problem in a heterogeneous energy-constrained environment is studied in [27]. Tasks in this system contain individual deadlines, and the goal is to complete as many tasks by their individual deadlines as possible within an energy constraint. This is a different problem from our work as we are trying to maximize the utility earned (based on each task's completion time) and not the number of tasks that meet their hard deadlines. The concept of an energy filter is used in [27], and we build on that for a more complex filter that automatically adjusts its level of filtering.

VII. Conclusion

In this paper, we study the problem of maximizing utility in an oversubscribed heterogeneous computing environment while satisfying an energy constraint. We examine and compare the performance of multiple heuristics in different task management environments. Our results indicate that the polled environment provides significant benefit over the queued environment in a system like ours because it has the ability to quickly service newly-arrived tasks with high utility. The queued environment with its pending slot and virtual queues may be more useful in an environment where the time between mapping events is longer than the average task execution time. We design, implement, and analyze multiple versions of the First Come First Served heuristic that are commonly used in many real-world schedulers. We compare the performance of these heuristics with Last Come First Served heuristics and other smart heuristics and demonstrate the strength of these smart heuristics. We also design a novel energy filtering technique that can be used with any of the heuristics and can be readily deployed in any environment without the need for any off-line parameter tuning. The adaptive energy filtering technique improves the performance of almost all the heuristics by allowing the heuristics to distribute their consumption of the budgeted energy and earn more utility.

Possible directions for future research include: (1) making the resource allocation techniques robust to uncertainties such as stochastic task execution times, machine failures, etc., (2)

experimenting with other types of task management environments, and (3) considering parallel and dependent tasks scheduling.

ACKNOWLEDGMENTS

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, supported by the Extreme Scale Systems Center at ORNL, which is supported by the Department of Defense. Additional support was provided by a National Science Foundation Graduate Research Fellowship, and by NSF Grants CCF-1302693 and CCF-1252500. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research also used the CSU ISTeC Cray System supported by NSF Grant CNS-0923386. The authors thank Mark Oxley and Daniel Dauwe for their valuable comments.

REFERENCES

[1] America's data centers consuming and wasting growing amounts of energy. [Online]. Available: http://www.nrdc.org/energy/data-center-efficiency-assessment.asp

[2] D. J. Brown and C. Reams, "Toward energy-efficient computing," *Communications of the ACM*, vol. 53, no. 3, pp. 50–58, Mar. 2010.

[3] P. Bohrer, E. N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony, "The case for power management in web servers," in *Power Aware Computing*, ser. Series in Computer Science, R. Graybill and R. Melhem, Eds. Springer US, 2002.

[4] I. Rodero, J. Jaramillo, A. Quiroz, M. Parashar, F. Guim, and S. Poole, "Energy-efficient application-aware online provisioning for virtualized clouds and data centers," in *International Green Computing Conference*, Aug 2010, pp. 31–45.

[5] B. Khemka, R. Friese, S. Pasricha, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, S. Powers, M. Hilton, R. Rambharos, and S. Poole, "Utility maximizing dynamic resource management in an oversubscribed energy-constrained heterogeneous computing system," *Sustainable Computing: Informatics and Systems*, vol. 5, pp. 14–30, Mar. 2015.

[6] Advanced configuration and power interface specification. [Online]. Available: http://www.acpi.info/spec.htm

[7] B. Khemka, R. Friese, L. D. Briceño, H. J. Siegel, A. A. Maciejewski, G. A. Koenig, C. Groer, G. Okonski, M. M. Hilton, R. Rambharos, and S. Poole, "Utility functions and resource management in an oversubscribed heterogeneous computing environment," *IEEE Transactions on Computers*, accepted 2014, to appear.

[8] "Introducing Titan," Jun 2014. [Online]. Available: https://www.olcf.ornl.gov/titan/

[9] R. Friese, B. Khemka, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, S. Powers, M. Hilton, J. Rambharos, G. Okonski, and S. W. Poole, "An analysis framework for investigating the trade-offs between system performance and energy consumption in a heterogeneous computing environments," in *22nd Heterogeneity in Computing Workshop (HCW 2013), in the proceedings of the IPDPS 2013 Workshops & PhD Forum (IPDPSW)*, May 2013, pp. 19–30.

[10] H. Barada, S. M. Sait, and N. Baig, "Task matching and scheduling in heterogeneous systems using simulated evolution," in *10th Heterogeneous Computing Workshop (HCW 2001), in the proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, Apr. 2001, pp. 875–882.

[11] M. K. Dhodhi, I. Ahmad, and A. Yatama, "An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 62, no. 9, pp. 1338–1361, Sep. 2002.

[12] A. Ghafoor and J. Yang, "A distributed heterogeneous supercomputing management system," *IEEE Computer*, vol. 26, no. 6, pp. 78–86, June 1993.

[13] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *IEEE Concurrency*, vol. 6, no. 3, pp. 42–51, July 1998.

[14] A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C. Wang, "Heterogeneous computing: Challenges and opportunities," *IEEE Computer*, vol. 26, no. 6, pp. 18–27, June 1993.

[15] H. Singh and A. Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," in *5th Heterogeneous Computing Workshop (HCW '96)*, Apr. 1996, pp. 86–97.

[16] D. Xu, K. Nahrstedt, and D. Wichadakul, "QoS and contention-aware multi-resource reservation," *Cluster Computing*, vol. 4, no. 2, pp. 95–107, Apr. 2001.

[17] Colorado State University ISTeC Cray High Performance Computing Systems. [Online]. Available: http://istec.colostate.edu/activities/cray

[18] M. R. Gary and D. S. Johnson, *Computers and Intractability: A guide to the theory of NP-Completeness*. W. H. Freeman and Co., 1979.

[19] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 107–121, Nov. 1999.

[20] T. D. Braun, H. J. Siegel, N. Beck, L. Boloni, R. F. Freund, D. Hensgen, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, June 2001.

[21] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on non-identical processors," *Journal of the ACM*, vol. 24, no. 2, pp. 280–289, Apr. 1977.

[22] S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, and Sa. Ali, "Representing task and machine heterogeneities for heterogeneous computing systems," *Tamkang Journal of Science and Engineering, Special Issue, Invited*, vol. 3, no. 3, pp. 195–207, Nov. 2000.

[23] J.-F. Pineau, Y. Robert, and F. Vivien, "Energy-aware scheduling of bag-of-tasks applications on masterworker platforms," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 145–157, Feb. 2011.

[24] R. Friese, T. Brinks, C. Oliver, A. A. Maciejewski, H. J. Siegel, and S. Pasricha, "A machine-by-machine analysis of a bi-objective resource allocation problems," in *The 2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2013)*, July 2013, pp. 3–9.

[25] K. H. Kim, R. Buyya, and J. Kim, "Power aware scheduling of bag-of-tasks applications with deadline constraints on DVS-enabled clusters," in *IEEE/ACM International Symposium of Cluster Computing and the Grid (CCGrid 2007)*, 2007, pp. 541–548.

[26] H. Yu, B. Veeravalli, and Y. Ha, "Dynamic scheduling of imprecise-computation tasks in maximizing QoS under energy constraints for embedded systems," in *Asia and South Pacific Design Automation Conference (ASPDAC 2008)*, Mar. 2008, pp. 452–455.

[27] B. D. Young, J. Apodaca, L. D. Briceño, J. Smith, S. Pasricha, A. A. Maciejewski, H. J. Siegel, B. Khemka, S. Bahirat, A. Ramirez, and Y. Zou, "Deadline and energy constrained dynamic resource allocation in a heterogeneous computing environments," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 326–347, Feb. 2013.