Dynamic Resource Management for Parallel Tasks in an Oversubscribed Energy-Constrained Heterogeneous Environment

Dylan Machovec[1], Bhavesh Khemka[1], Sudeep Pasricha[1,2], Anthony A. Maciejewski[1], Howard Jay Siegel[1,2]
Gregory A. Koenig[3], Michael Wright[4], Marcia Hilton[4], Rejendra Rambharos[4], and Neena Imam[3]
[1]Department of Electrical and Computer Engineering
[2]Department of Computer Science
Colorado State University
Fort Collins, CO 80523, USA
[3]Oak Ridge National Laboratory
Oak Ridge, TN 37831, USA
[4]Department of Defense
Washington, DC 20001, USA
djmachov@rams.colostate.edu, bhavesh.khemka@gmail.com, {sudeep,aam,hj}@colostate.edu,
koenig@ornl.gov, michael.wright4@comcast.net, mmskizig@verizon.net, jendra.rambharos@gmail.com, imamn@ornl.gov

***Abstract*—The worth of completing parallel tasks is modeled using utility functions, which monotonically-decrease with time and represent the importance and urgency of a task. These functions define the utility earned by a task at the time of its completion. The performance of such a system is measured as the total utility earned by all completed tasks over some interval of time (e.g., 24 hours). To maximize system performance when scheduling dynamically arriving parallel tasks onto a high performance computing (HPC) system that is oversubscribed and energy-constrained, we have designed, analyzed, and compared different heuristic techniques. Four utility-aware heuristics (i.e., Max Utility, Max Utility-per-Time, Max Utility-per-Resource, and Max Utility-per-Energy), three FCFS-based heuristics (Conservative Backfilling, EASY Backfilling, and FCFS with Multiple Queues), and a Random heuristic were examined in this study. A technique that is often used with the FCFS-based heuristics is the concept of a permanent reservation. We compare the performance of permanent reservations with temporary place-holders to demonstrate the advantages that place-holders can provide. We also present a novel energy filtering technique that constrains the maximum energy-per-resource used by each task. We conducted a simulation study to evaluate the performance of these heuristics and techniques in an energy-constrained oversubscribed HPC environment. With place-holders, energy filtering, and dropping tasks with low potential utility, our utility-aware heuristics are able to significantly outperform the existing FCFS-based techniques.**

***Keywords*—heterogeneous computing; energy-aware computing; utility functions; resource management heuristics; parallel tasks; scheduling**

I. Introduction

High performance computing (HPC) environments are commonly used to execute computationally intensive tasks. These tasks are often parallel, meaning that they utilize multiple cores within an HPC environment to reduce their execution time. It is necessary to have resource managers that execute the workload arriving into the system in a way that attempts to maximize the amount of useful work that the system accomplishes. This is especially important when the system is oversubscribed, i.e., the system cannot execute each task as soon as it arrives in the system.

The heterogeneous HPC environments that we modeled in this study are based on those being investigated by the Extreme Scale Systems Center (ESSC) at Oak Ridge National Laboratory (ORNL). The ESSC is part of a collaborative effort between the Department of Energy (DOE) and the Department of Defense (DoD) to perform research and deliver tools, software, and technologies that can be integrated, deployed, and used in HPC environments in both DOE and DoD.

Many systems use metrics such as "utilization" of machines or "fairness" to measure the performance of the resource manager. Because we consider an oversubscribed heterogeneous environment, utilization is not an effective performance measure. This is because assigning a task to the node types that take the longest to complete the task (i.e., not types that are not effective for that task) will result in higher system utilization for that task. Furthermore, because the system is oversubscribed, we would expect to always have near 100% utilization. Because different tasks have different importance to the users of this environment, fairness (i.e., equal treatment of all tasks) should not be used.

To effectively model the performance of an oversubscribed heterogeneous system, we employ the concept of utility functions [9], which are monotonically-decreasing functions of time that represent the importance and urgency of

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory (ORNL), supported by the Extreme Scale Systems Center at ORNL, which is supported by the Department of Defense (DoD); and by NSF Grant CCF-1302693. This work also utilized CSU's ISTeC Cray system, which is supported by the National Science Foundation (NSF) under grant number CNS-0923386.

978-1-5090-3682-0/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.25

a task and define the utility earned by a task at the time of its completion. The performance of the system is measured by the total utility earned from completing tasks in a given period of time. We refer to this as the system utility.

Electric energy is an expensive and potentially limited resource (e.g., [2, 21]). In this study, we constrain the amount of energy that the HPC system can consume each day. To effectively maximize system utility while satisfying this energy constraint, heuristics are needed. This is because the general problem of mapping tasks onto a set of resources is known to be NP-hard [5]. It is not possible for an algorithm to find optimal solutions to NP-hard problems for a realistic system in a reasonable amount of time. We utilize energy filtering techniques to improve the energy efficiency of our heuristics.

We design four utility-aware resource allocation heuristics: Max Utility, Max Utility-per-Time, Max Utility-per-Resource, and Max Utility-per-Energy. We compare these to four approaches from the literature: Conservative Backfilling, EASY Backfilling, FCFS with Multiple Queues, and Random [11, 15, 18].

Many heuristics for the resource allocation of parallel tasks in HPC environments use permanent reservations to allow for allocations of nodes to tasks in the future (e.g., [15, 18]). Because permanent reservations that are never moved can be restrictive, we made use of the concept of place-holders, which we previously developed in [11]. This provides additional flexibility by allowing newly arriving tasks of high utility to replace tasks that have reserved resources with place-holders.

The novel contributions of this work are:

- The design of an energy-per-resource filtering technique and utility-aware heuristics with the goal of maximizing utility earned by parallel tasks while obeying an energy constraint in heterogeneous oversubscribed HPC environments;
- The creation of a parallel simulator that can model various HPC environments based on those of interest to the ESSC at ORNL;
- The use of the simulator to analyze and evaluate these new heuristics, and compare them to the existing permanent-reservation-based heuristics and techniques in terms of both utility earned and energy efficiency.

This paper is organized as follows. In Section II, we define the environment and problem. Section III explains the resource management techniques that are utilized. The setup for our simulator is detailed in Section IV. The simulation results are presented in Section V. In Section VI, we discuss related work. Finally, in Section VI we conclude and discuss potential future work.

II. Environment and Problem Description

A. Compute System Model

We modeled an environment where the compute system is composed of heterogeneous clusters of nodes, as shown in Figure 1. A node is the atomic unit of resource allocation in this model. Each node is composed of one or more multicore processors (MCPs). The nodes that form each cluster are homogeneous, meaning that they are identical (and therefore have the same number and type of MCPs). The node architecture varies across clusters and there can be different numbers of MCPs per node across clusters. We modeled MCPs that utilize dynamic voltage and frequency scaling (DVFS) to switch among multiple performance states (P-states), where each P-state provides different power consumption and execution speed [6].

B. Workload and Environment Characteristics

Tasks arrive dynamically, are of different types, and may be designed for parallel execution, which means that they may be required to execute on multiple nodes concurrently. The environment is oversubscribed, that is, it is not possible for all tasks to earn their maximum utility because of delayed completion time. In this study, we do not allow a task to be assigned across nodes in separate clusters because the nodes have different architectures. Our model assumes that tasks do not communicate with one another because they are independent and potentially submitted by different users. We make the assumption that tasks cannot be preempted (i.e., once they start, they execute to completion).

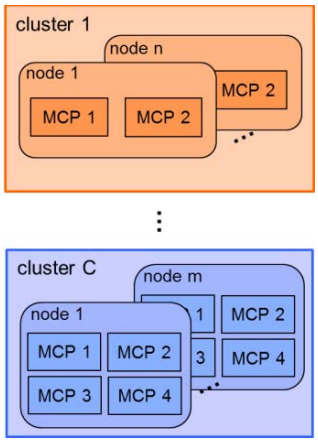


Figure 1. A possible compute system composed of C clusters. Cluster 1 has n nodes each with two MCPs and cluster C has m nodes each with four MCPs.

Task types in this environment have execution characteristics (execution time and energy consumed) that are deterministic and known to the system's resource manager. We assumed that this information would be available through historical and experimental data. Tasks with the same execution characteristics belong to the same task type. Whenever a task arrives it specifies its task type, the number of nodes that it will need depending on the cluster it is assigned to, and its utility function. Because the environment is heterogeneous, cluster A may be faster (or more energy efficient) than cluster B for one task type, but not for all other task types.

Within each cluster, execution characteristics are defined by an Estimated Time to Compute (ETC) matrix and an Average Power Consumption (APC) matrix [19]. The ETC matrix is used to specify the execution time of tasks for any task type, cluster, and P-state combination for some number of nodes. Because nodes within a cluster are homogeneous, the ETC only needs to reference the number of nodes that a task type will use in a given cluster. An example of part of an ETC matrix, where the cluster and P-state have already been

selected, is shown in Figure 2. It is assumed that from past executions or experiments we have entries for certain levels of parallelism, i.e., for certain numbers of nodes. In our simulations, if the number of nodes the task needs is not listed in the ETC then its execution time is assumed to be between two values listed in the matrix. We then calculate its execution time using linear interpolation. In some cases a task type's execution time may increase (instead of decrease) with an increased number of nodes due to communication and synchronization overheads.

task type	number of nodes above execution time				
1	1	2	4	16	32
	100	70	50	25	30
2	256	512			
	300	200			
3	8	16	64		
	100	80	70		

Figure 2. An example of an ETC matrix that specifies execution time for task type and number of nodes for a given cluster and P-state.

The APC matrix defines the average power consumption of the nodes that a task will utilize and is structured similarly to the ETC matrix. We can calculate an estimate of the total energy that any task will consume by multiplying its execution time and average power consumption value.

In Figure 3, the interaction between the different components of the modeled system is shown. Tasks arrive dynamically and are sent to the resource manager. The resource manager will use the ETC and APC information, in addition to the task's utility function, to map the tasks to nodes in one of the clusters.

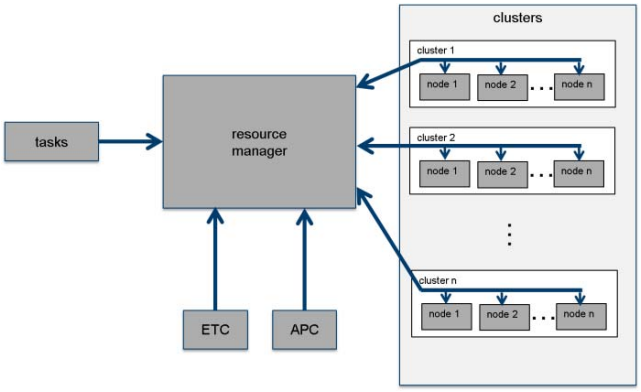


Figure 3. Flow for the proposed resource manager. Tasks enter the resource manager and are mapped to the nodes of clusters. Each task is mapped to the nodes of a single cluster.

C. Utility Functions

Our performance metric is based on a very flexible measure of the importance of a task that depends on when the task is completed. Utility functions are monotonically decreasing functions that define the utility that a task earns upon completion, depending on the amount of time that has passed since the task was submitted to the system (see Figure 4). In this study, utility functions are defined by three parameters: priority, urgency, and a utility class. The priority of a utility function is equal to its starting utility (the maximum it can possibly earn). Urgency is used to define the rate at which the utility function will decay. The utility for functions with a higher urgency value will decrease at a faster rate than those with a lower value. The utility class defines the shape of the utility function, and is scaled using the priority and urgency. We define utility functions as piecewise exponential functions. For each exponential interval after the first, the utility class contains the start time of the interval, a percentage of the starting utility, and an urgency modifier. Each task has an associated utility function that may be distinct from the utility functions of other tasks. The way in which utility functions can be created is discussed with more detail in our previous work [9]. In general use, a utility function can be simplified by specifying it as a linear representation.

D. Problem Statement

We define the system utility earned over some interval of time as the sum of utility earned by tasks that are completed by the system during that interval. This also includes a portion of the utility earned by each task if the task would be partially completed during that time interval. For example, if a task were to complete 70% of its total execution time during interval A and 30% of its total execution time during interval B, then the utility earned for this task during interval A would be 70% of the task's final utility and the utility earned during interval B would be the remaining 30% of the task's final utility. The system is oversubscribed, and has an energy constraint, which is the maximum amount of energy that it can consume during some time interval. *The goal of our resource manager is to maximize the utility earned by the system while obeying the energy constraint.*

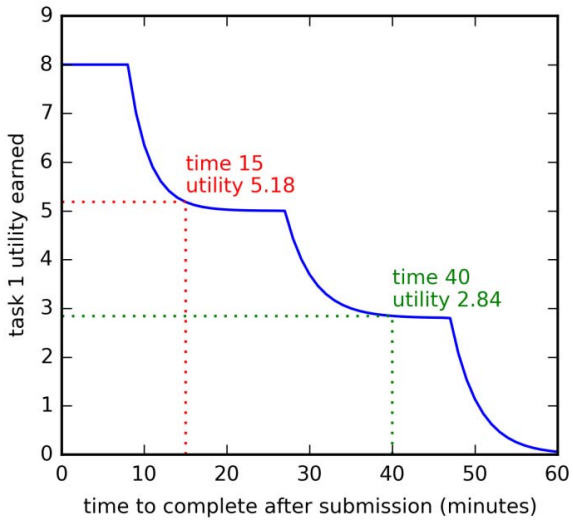


Figure 4. An example of a utility function for task 1. If task 1 completes at time 15, it earns 5.18 utility. If task 1 complete at time 40, it earns 2.84 utility.

III. Resource Management

A. Mapping Events

Mapping is the process of assigning and scheduling tasks to the nodes of the HPC system. When a task arrives in the system, it is added to the set of mappable tasks. Once a task is mapped to nodes, it is removed from this set. During a mapping event, the resource manager makes allocation decisions for some or all mappable tasks in the system. In each mapping event, three techniques are used. First, some of the tasks are dropped to tolerate oversubscription (described in Subsection III.C). Next, energy filtering (detailed in Subsection III.G) attempts to improve energy efficient by limiting the allocation options for tasks. Finally, one of the heuristics defined in Subsection III.D or III.E is used to make the final resource management decisions. In the environment we simulated, mapping events occur every 60 seconds, but this can be changed depending on factors such as task arrival rates and the average execution time of tasks.

B. Permanent Reservations and Place-holders

A permanent reservation marks the resources that will be allocated to a task at some point in the future. Throughout this paper, we refer to the *resources allocated* to a task as the amount of time that the task will take to execute multiplied by the number of cores that are allocated to that task:

$$\textit{resources allocated} = \textit{execution time} \times \textit{cores allocated}. \quad (1)$$

The number of *cores allocated* to a task is equal to the number of nodes that are allocated to the task multiplied by the number of cores on each node. Permanent reservations cannot be removed or moved, i.e., they ensure that the reserved task will start execution on those resources at that future time. A place-holder, proposed by us in [11], is similar to a permanent reservation, except that all place-holders are removed from the system at the beginning of the next mapping event. This creates opportunities for newly arriving tasks that may earn more utility than the tasks that would be executed instead if permanent reservations were used.

If a task cannot begin executing immediately on available nodes, a permanent reservation or place-holder can be made for the task so that it can begin executing at a future time. This is done so that the resource manager is aware of tasks that cannot begin executing immediately due to required resources being unavailable.

C. Task Dropping

At the start of a mapping event, we calculate the amount of utility that each task can earn if it were to start execution immediately in the cluster that has the minimum execution time for the task. This amount is an upper bound on the utility that the task can earn and may be more than is realistically achievable. If this amount of utility is lower than a preset dropping threshold, then the task is dropped from the system. This is done so that tasks that are unable to earn a significant amount of utility are removed from the set of mappable tasks. This is particularly important when dealing with permanent reservations because without dropping, reservations for tasks that earn little utility can be made far into the future. This can be a poor use of the system's resources in terms of earning little utility for the reserved task, and can also result in reduced utility being earned for the tasks that arrive in the future due to the delay caused by these reservations. The dropping threshold may be set by simulation experiments.

D. Comparison Heuristics

Four of the heuristics we simulated were for the purpose of comparison to our utility-aware resource management techniques. These heuristics are used in parallel scheduling literature.

1) Random: This heuristic takes the tasks in order of arrival, and assigns a task to a random cluster with a random P-state. This process is repeated until all mappable tasks have been assigned to some cluster, or until no more assignments are possible.

2) Conservative Backfilling: This heuristic, described in [18], considers tasks in order of arrival and assigns each task to a cluster where it can start execution immediately. If there is no cluster where the task can start execution, then the heuristic makes a permanent reservation for the task on a cluster where the task can start execution as soon as possible. This process is repeated until each task is executing or has a permanent reservation, or it is not possible to assign any more tasks to the system because there is no availability in the system to start the task before the end of the day being simulated. This heuristic employs backfilling, the process of assigning tasks to the voids (gaps in node usage) in the schedule that can occur when reservations are made for a future time. A backfilled task may be able to start execution immediately or may create another reservation. This heuristic was designed for homogeneous clusters and an environment that does not consider utility or energy. The heuristic always used P-state 0.

3) EASY Backfilling: Extensible Argonne Scheduling sYstem (EASY) Backfilling, from [15], is a common heuristic for scheduling parallel tasks. It initially works in the same way as Conservative Backfilling. The major difference is in how it handles permanent reservations. It makes a permanent reservation for a single task that cannot start execution immediately such that the task will start execution as soon as possible, but will not make reservations for any tasks if a reservation already exists. Similar to Conservative Backfilling, it will still continue to search for backfilling opportunities for other tasks, as long as they can start execution immediately without delaying the single reservation. This heuristic was designed for homogeneous clusters and an environment that does not consider utility or energy. The heuristic always used P-state 0.

4) FCFS with Multiple Queues: The FCFS (first come, first served) with multiple queues heuristic, used in [11], is another comparison heuristic. It is similar to the Conservative Backfilling heuristic, except that it uses multiple queues instead of a single FCFS queue. The purpose of these queues is to separate the tasks based on their expected resource usage. The three queues used in this study are labeled small, medium, and large. Based on the information available in the ETC matrix, it is possible to determine the amount of resources that

any task will be allocated, averaged over the clusters. Once this average is found, we then use the average resources that a task is allocated when scheduled to determine onto which queue that task will be appended. The tasks are added to queues in order of their arrival. Tasks that consume less than a lower threshold of resources will be added to the "small" queue. Tasks that consume more than an upper threshold of resources will be added to the "large" queue. The rest will added to the "medium" queue. In our simulation study, the lower threshold was set to 30% of the average resources of the task that needs the most resources in the system and the higher threshold was set to 60% of the average. The rest of the execution of the heuristic is identical to Conservative Backfilling, except that instead of taking one task at a time from the single queue, the heuristic will cycle through the three queues in a round robin manner such that within each cycle, at most one large task is assigned, at most four medium tasks are assigned, and finally at most eight small tasks are assigned. The lower threshold, upper threshold, and the number of tasks selected from each queue in each iteration are examples of what may be used in a system, but different systems may use different values. The motivation for this heuristic is to attempt to balance the tasks being assigned to the system based on resources needed.

E. Utility-Aware Heuristics

1) Overview: We have designed four utility-aware heuristics that can be used with permanent reservations or with place-holders. All of these heuristics use a framework that is based on the concept of the Min-Min scheduling technique from [7], which has been used successfully in many environments (e.g., [3, 16]), but none with the same set of conditions as we have here. All of these utility-aware heuristics have a similar structure that defines their execution, but utilize a different objective measure.

2) Heuristic Objective Measures: We utilize four objective measures for our heuristics. These are, Utility (Util), Utility-per-Time (UPT), Utility-per-Resource (UPR), and Utility-per-Energy (UPE):

$$Util = \text{value of the task's utility function at completion}, \quad (2)$$

$$UPT = Util / \text{the task's execution time}, \quad (3)$$

$$UPR = Util / \text{resources allocated to the task}, \quad (4)$$

$$UPE = Util / \text{energy consumed by the task}. \quad (5)$$

We define four heuristics, Max Util, Max UPT, Max UPR, and Max UPE using the objective measures listed in (2)-(5). Max Util and Max UPR were used in a parallel environment in our work in [11], but in that work: (a) there was no consideration of energy at all (thus, there were no energy filters with Max Util and Max UPR, and the Max UPE heuristic was not considered), (b) the performance measure was based on linear "value functions" with hard and soft deadlines rather than the more flexible utility functions used here, and was not constrained by an energy budget, and (c) the computing system and workload environment were very different, and were not based on typical DOE and DoD environments. Max Util, Max UPT, and Max UPE were used in our work in [9, 10] with an energy constraint, but only for serial tasks. Thus, this work is significantly different from our work in [9, 10, 11] because the heuristics are designed for parallel tasks and must obey an energy constraint, and we assume a different performance metric and environment.

3) Maximizing the Objective Measure for Each Task: The first phase of these heuristics involves finding the maximum value of the heuristic's objective measure for each mappable task. This is done by selecting an allocation of nodes within each cluster that maximizes this objective measure (varying the P-state as needed to achieve this maximum). This is shown for the Max UPE heuristic in Algorithm 1, lines 2 to 3.

4) Assigning Tasks to Resources: Once a maximum objective measure allocation has been found for each unmapped task, the task that has the highest maximum objective measure is assigned to its selected resources (defined by nodes, a P-state, a start time, and a finish time). This may create a permanent reservation if necessary (i.e., when the task cannot start execution at the current time). The task is removed from the set of mappable tasks. This process of greedily assigning tasks to resources is repeated until no more unmapped tasks exist in the system, or until it is not possible to assign any more tasks (due to running out of energy or reaching the end of the day). This is shown in Algorithm 1 for the Max UPE heuristic in lines 4 to 10. This algorithm can also use place-holders instead of permanent reservations, by replacing "permanent reservations" with "place-holders" in line 9 of Algorithm 1.

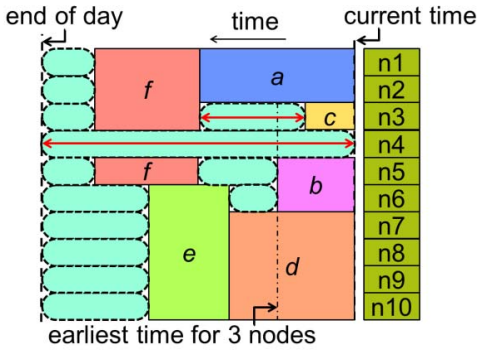
Algorithm 1. Pseudo-Code for the Max UPE Heuristic

```
1.  while mappable tasks is not empty and a mappable task exists
    that can be scheduled to begin executing during the current day
    based on energy remaining
2.      for each task in mappable tasks
3.          find nodes/cluster/P-state combination that maximizes
            UPE for the task
4.      select task from mappable tasks with cluster/P-state
        combination that has the highest maximum UPE
5.      if selected task can start execution immediately with that
        nodes/cluster/P-state combination
6.      then
7.          assign selected task to that cluster/P-state combination
8.      else
9.          create a permanent reservation for selected task on
            that nodes/cluster/P-state combination
10.     remove task from mappable tasks
11. end while
```

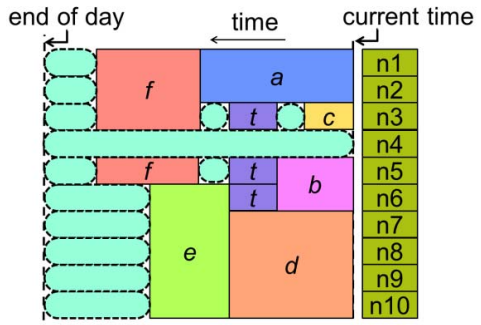
F. Finding Allocation Options for a Task

We adapt a technique from our work in [11] to find the node allocation for a task within a cluster (line 3 of Algorithm 1). Given that nodes in a cluster are homogeneous, the maximum value of any heuristic's objective function for each task/P-state combination (e.g., UPE) within each cluster is

achieved when that task finishes execution as soon as possible in that cluster with that P-state. A task's execution time will be the same irrespective of which nodes in a cluster it uses. Because of this, finding the earliest possible finish time for a task is equivalent to finding the task's earliest possible start time. The allocation options that are considered for a task are the earliest possible start time for each P-state/cluster combination. This strategy is used for all twelve heuristics that we present in this paper (i.e., the comparison heuristics discussed in Subsection 3.D and the utility-aware heuristics detailed in Subsection 3.E).



(a) State of a cluster before assigning task *t*



(b) State of the cluster with task *t* assigned

Figure 5. An example of a mapping on a cluster with ten nodes, based on our work in [11]. The colored rectangles represent different tasks, and the rounded rectangles represent voids where new tasks can be inserted. In (a), the earliest time when three nodes are available is shown. In (b), the three nodes chosen for task *t* are shown. Note that *n4* is not chosen due to the second tiebreaking criteria described in Subsection III.F.

When the earliest possible starting time for a task is found within a cluster, it is possible that there will be a set of nodes to choose from that contains more nodes than are requested by the task. In this case, we use two criteria to attempt to pick the best subset of nodes. The first of these criteria is to pick nodes that cause the smallest number of idle voids in the system (i.e., sections of time between the executions of two tasks on node). The second criterion is to compare the size of the idle voids into which the task would be inserted, and to choose the nodes with the smallest voids. An example of this process is shown in Figure 5, where a task *t* is requesting three nodes. The earliest time when three nodes are available is a time when four nodes are available and we use this algorithm to select three nodes out of the four. In this example, *n6* and *n5* are selected in that order using the first criterion, and then *n3* is selected using the second criterion. The node *n4* is not selected. The motivation for these criteria is to reduce the overall fragmentation of the schedule to give future tasks a better chance of being backfilled.

G. Energy Filtering

1) Overview: We have designed energy filtering techniques to improve the effectiveness of our utility-aware heuristics under an energy constraint. An energy filter is used to remove allocation options that exceed a notion of "fair share" of energy consumption. The motivation for energy filtering is to limit the rate at which energy is consumed by the resource manager until the energy constraint is reached at the end of the day. Without energy filtering, many heuristics will use up their energy part way through the day resulting in lost utility due to the inability to execute potentially high utility tasks that arrive after the energy constraint has been reached.

2) Energy-per-Task Filtering: We calculate the energy-per-task budget for a task as the fair share of energy that that task is permitted to consume. This budget, extended from the serial version of this energy filter in [10] to apply to parallel tasks, is calculated using the energy remaining in some interval of time (such as a day), and the estimated number of tasks that will be executed in that same interval:

resources remaining =

$$\sum_i \frac{unallocated\ time\ remaining\ on\ node\ i}{\times\ cores\ in\ node\ i}, \quad (6)$$

estimated number of tasks remaining =

$$min\left(\frac{resources\ remaining}{average\ resources\ used}, \frac{energy\ remaining}{average\ energy\ consumed}\right), \quad (7)$$

energy-per-task budget =

$$leniency\ factor \times \frac{energy\ remaining}{estimated\ number\ of\ tasks\ remaining}. \quad (8)$$

In addition, a leniency factor is included that can be set by simulation experiments to improve the performance of the filter. Any task allocation options where the task would consume energy greater than the energy budget are not considered by heuristics.

3) Energy-per-Resource Filtering: In our previous work, there were serial tasks that caused a one-to-one mapping between a single task and a single resource. In contrast, here we are considering parallel tasks that can use different numbers of resources. We need to consider the resources needed when designing the energy filter. We present a new energy-per-resource filter that provides better performance in an environment with parallel tasks. We calculate the energy-per-resource budget as the fair share of energy-per-resource that a task is permitted to consume. This energy-per-resource budget is calculated by dividing the energy remaining in the

day by the unallocated resources remaining in the system during the day:

energy-per-resource budget =

$$leniency\ factor \times \frac{energy\ remaining}{resources\ remaining}. \quad (9)$$

Again, we multiply by a leniency factor determined by simulation experiments to improve the results of this filter. We can then calculate the energy-per-resource of any task allocation as the amount of energy that the allocation will consume, divided by the resources allocated to the task, as defined in (1). If the energy-per-resource of some task allocation exceeds the energy-per-resource budget, then that allocation is not considered by the heuristics.

IV. Simulation Setup

A. Overview

The simulation setup described in this section was designed based on discussions with researchers from ORNL and DoD. We generate 48 simulation trials as described in Subsection V.B. We simulate a total of 28 hours, but only analyze the results for the last 24 hours of each simulation. The first four hours ensure that the simulated system does not begin with all nodes in an idle state.

B. Generation of Compute System and Workload

The compute system we simulate is composed of 100,000 cores that are distributed across six heterogeneous clusters. Of the clusters, four are general-purpose and two are special-purpose. It is assumed that each special-purpose cluster will have more cores on average than the individual general-purpose clusters. The difference between general-purpose and special-purpose clusters is the type of tasks they are able to execute.

In our experiments, two workloads are considered. The first has a mean of 5,000 tasks arriving per day and the other has a mean of 10,000 tasks arriving per day. Each of the tasks belongs to one of 100 task types that we generate for each simulation trial, i.e., with different ETC values and APC values. Of these 100 task types, 60 are general-purpose tasks types and 40 are special-purpose tasks types. General-purpose tasks can only run on the general-purpose clusters. Special-purpose tasks can only run on their special-purpose cluster. Of the special-purpose task types, 20 are limited to each special-purpose cluster.

The execution time for each task type on a single core is sampled from a Gaussian distribution with a mean that is between 1 hour and 18 hours. This mean is selected using the starting utility that the task earns. This is done because there is a correlation between a task's single core execution time and its starting utility in our simulation study. In addition, the distribution of task types in our environment is defined by the priority level of each task type (shown in Table I).

To describe a task's utility function, we use three parameters: priority, urgency, and utility class [9]. The priority (or starting utility) and urgency parameter for the utility function of each task type are generated using the distribution of priority and urgency shown in Table I (from [9]). The actual starting utility value is chosen uniformly from the utility range associated with each priority level. For each of the tasks, a utility class (defined in Subsection II.C) is randomly selected from one of 20 that we have generated for our simulation studies.

TABLE I. PRIORITY AND URGENCY TABLE

priority level	utility range	urgency rate			
		0.6	0.2	0.1	0.01
critical	(6,8]	2%	2%	0.05%	0%
high	(4,6]	3.45%	5%	1.5%	3%
medium	(2,4]	0%	10%	10%	10%
low	[1,2]	0%	0%	20%	33%

For our simulation studies, we assume there is a correlation between the single core execution time of a task and the starting utility value it has. This is based on the assumption that in our intended environment longer running tasks are generally of higher importance.

The perfectly correlated values for starting utility (priority) and single core execution time are defined such that task types with the minimum possible single core execution time (set to 1 hour) have the minimum possible starting utility (set to 1). Similarly, task types with the maximum possible single core execution time (set to 18 hours) would have a starting utility of the maximum starting utility (set to 8). These perfectly correlated end points are used as the mean values for the Gaussian distribution described above for determining the single core execution time for each task. The correlation between the single core execution time of a task type and the starting utility of a task that we use for the 100 task types in the 48 simulation scenarios can be seen in Figure 6. This correlation is generated by using a coefficient of variation (COV) value of 0.15 for the Gaussian distribution described above. The execution time on other clusters (i.e., the heterogeneity) is modeled using the COV method from [1] with a COV parameter of 0.3. The entries of the APC matrix are also generated using the COV method for generating ETC matrices [1]. The mean power consumption is 133 watts for a single node and a COV of 0.2 is used in the method.

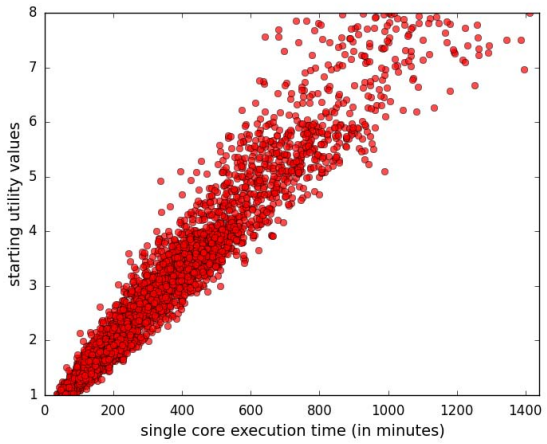


Figure 6. The correlation between single core execution time and starting utility for the 48 simulation scenarios each with 100 task types, for a total of 4800 points.

Once we have the complete set of task types, individual tasks must be generated for each task type. We define a mean

number of total tasks to generate an equal mean number of tasks for each task type. From this mean number of tasks, the mean rate of task arrival is calculated by dividing the mean number of tasks of each type by the duration of a day (24 hours). We sample the uniform distributions created by Table II to determine that the number of cores that each task of a task type will use. The values in this table, which were used for our simulation study, were based on typical DOE and DoD environments. Next, we generate the arrival pattern for a task type. If the task type requires less than or equal to 4096 cores, then its tasks will arrive with a sinusoidal pattern throughout the 24-hour period. All other tasks will instead arrive with a high rate during work hours (i.e., between 9:00 AM and 6:00 PM) and a low rate at other times during the day. This is done to model the expected arrival patterns for workloads of interest to DOE and DoD. The high rate is equal to two times the mean rate of the task type and the low rate is set below the mean rate so that the average arrival rate over the day is still equal to the mean rate. The execution time for parallel tasks in our simulations is determined from the single core execution times using the Downey model for the speedup of parallel programs [4]. The Downey sigma value used is sampled uniformly between 4 and 10.

TABLE II. CORE DISTRIBUTION OF TASKS

percentage of tasks	minimum cores	maximum cores
20%	2	4
20%	5	256
40%	257	4096
19%	4097	max cores of cluster – 1
1%	max cores of cluster	max cores of cluster

Cores may have many P-states. For our simulation study, we assume they each have three P-states. This provides for a choice between the lowest P-state, the highest P-state, and an intermediate P-state. This allows for energy-aware heuristics and techniques to improve energy efficiency while keeping the search space for allocations tractable. All cores in each node must always have the same active P-state. For our simulations, the differences among P-states for the same node type are defined using a power scaling factor. This factor is used to determine by what fraction the average power usage and execution time of each task will be adjusted. The three P-states have power scaling factors of 1.0, 0.75, and 0.5. A randomness factor is generated by sampling a gamma distribution with a mean of 1, and a COV of 0.3 for general-purpose tasks and 0.2 for special-purpose tasks. The adjustments to power consumption for the three P-states on all nodes are sampled from three other gamma distributions with means equal to the power scaling factor multiplied by the randomness factor with a COV of 0.03 for general-purpose tasks and 0.02 for special-purpose tasks. The execution time scaling for each P-state is sampled similarly to the power scaling, except that the execution time scaling is equal to the reciprocal of a sample from a gamma distribution that has a mean equal to the square root of the power scaling factor multiplied by the randomness factor [10].

C. Resource Management Parameters

1) Dropping Threshold: The dropping threshold for our resource manager is set to 0.5. This means that tasks that can no longer earn utility greater than 0.5 if they were to start execution immediately in their fastest cluster will be dropped from the system. We selected this threshold value because it gives all tasks the opportunity to execute. (i.e., because all tasks arrive with a starting utility of at least 1.0, it is possible for them to be mapped to nodes in the system). We used the same dropping threshold in our previous work [9, 10]. Lower dropping thresholds resulted in all heuristics earning less or equal utility than they did with a dropping threshold of 0.5. All utility-based heuristics made use of the utility dropping operation in this study. The FCFS-based heuristics and Random did not use this technique because it is not used with these heuristics in the literature. In actual practice, the threshold can be set based on simulation experiments modeling the real system environment to be used.

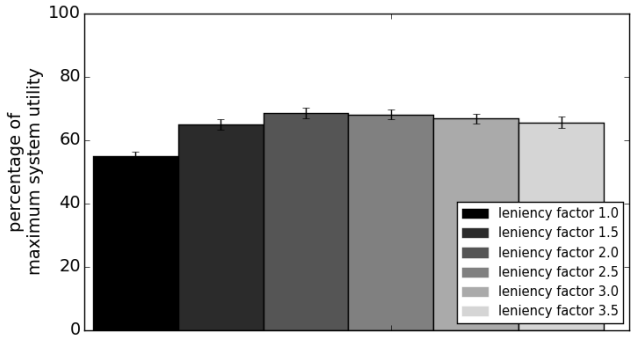


Figure 7. A range of energy leniency factors using energy-per-task filtering for the Max UPR with place-holders heuristic.

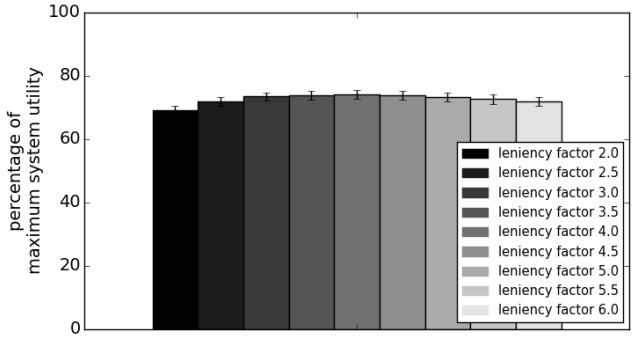


Figure 8. A range of energy leniency factors using energy-per-resource filtering for the Max UPR with place-holders heuristic.

2) Energy Filter Leniency Factors: We define the maximum system utility as the utility that would be earned if all tasks began execution at the time that they were submitted to the system. This is an upper bound on how much utility can be earned, i.e., the system utility, but is unobtainable in an oversubscribed environment because by definition all tasks cannot earn their individual maximum utility values (as discussed in Subsection II.C). The leniency factors for the two energy filters were both selected empirically by simulation experiments, by varying the energy leniency factor for the Max UPR heuristic with place-holders as seen in Figures 7

and 8 for a mean of 5,000 tasks arriving per day The confidence intervals are based on the 48 simulation trials. The leniency factor that performed the best was then used for all utility-based heuristics that were not energy aware (i.e., Max Util, Max UPT, and Max UPR) with permanent reservations and with place-holders. Using these results, a leniency factor of 2.0 was chosen for the energy-per-task filter and a leniency factor of 4.0 was chosen for the energy-per-resource filter. The FCFS heuristics from the literature did not make use of energy filtering in this study because this is not an operation that is used with these heuristics. Max UPE with permanent reservations and with place-holders also does not use the energy filter because it performs the best in terms of utility earned when an energy filter does not remove some of its resource allocation options. The energy leniency factors for a mean of 10,000 task arrivals were determined using the same method. As for the dropping threshold, in actual practice, the leniency factors can be set based on simulations experiments modeling the real system environment to be used.

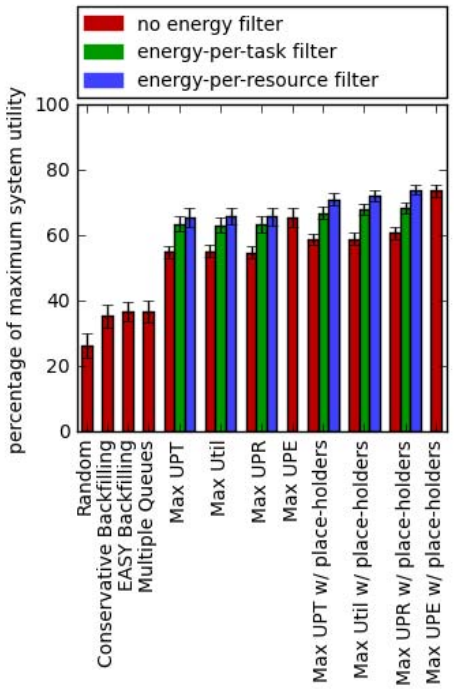
3) Energy Constraint: We set the energy constraint by running simulations without an energy constraint and observing how much energy the best heuristics consumed. We then set the energy constraint to a fraction of the energy that the best heuristic consumed to show the advantages of the utility-aware approaches.

The energy constraint for our simulations for a mean of 5,000 tasks arriving was set to 70% of the energy consumption for the Max Util with place-holders heuristic (and no energy constraint) because this heuristic earned the highest mean percentage of maximum utility. This resulted in an energy constraint of 12 billion joules.

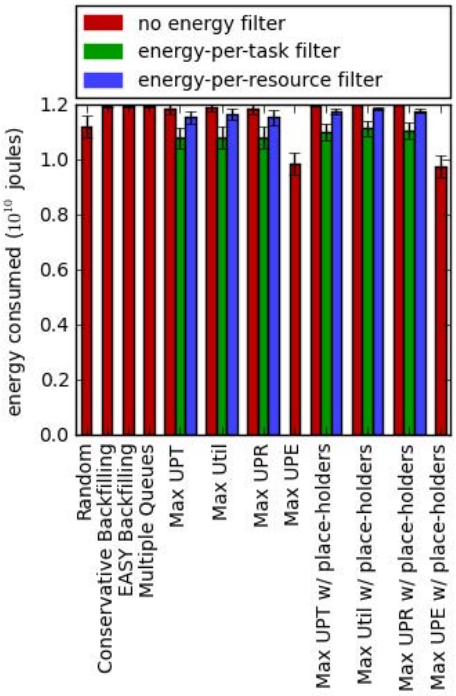
When the level of oversubscription of the system was increased by modeling a mean of 10,000 task arrivals per day, and there was no energy constraint, Max UPR with place-holders was the best heuristic. The energy constraint for a mean of 10,000 tasks arriving per day was set to 70% of the energy consumed by the Max UPR with place-holders heuristics equal to 15 billion joules. In a real system, the energy constraint would be set by the system administrator.

V. Results

In Figure 9(a), the percentage of maximum system utility earned in an energy constrained environment for a mean of 5,000 tasks is shown. Here, the utility-based heuristics made use of task dropping. In addition, results are shown for simulations where the utility-based heuristics used no energy filtering, energy-per-task filtering, and energy-per-resource filtering. The energy consumption for these results can be seen in Figure 9(b) with an energy constraint of 12 billion joules. Using either energy filtering technique allows the utility-based heuristics that are not energy aware to operate with a higher level of energy efficiency. This allows them to earn significantly more utility than they did when an energy constraint was set with no energy filtering. The confidence intervals in Figure 9 are based on the 48 simulation trials.



(a) percentage of maximum utility for a mean of 5,000 task arrivals



(b) energy consumption for a mean of 5,000 task arrivals

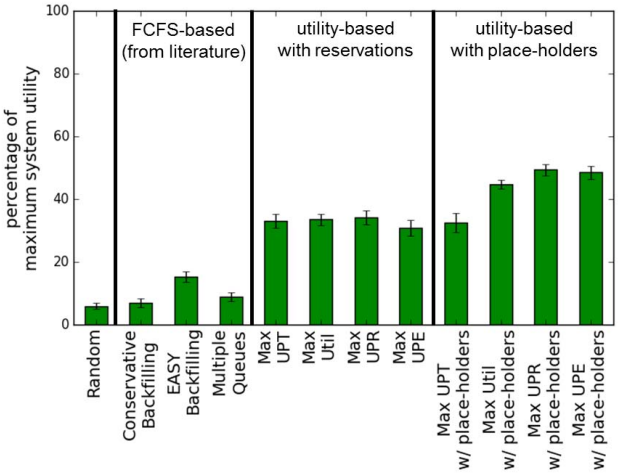
Figure 9. Results for a mean of 5,000 tasks arriving per day. The utility-based heuristics utilize task dropping with a dropping threshold of 0.5, and the utility-based heuristics that are not energy aware are also shown with and without the energy-per-task and energy-per-resource filters. In (a), a comparison of the percentage of maximum utility earned with 95% confidence intervals is shown. The energy consumption of each heuristic can be seen in (b) with 95% confidence intervals.

The utility-aware heuristics (Max Util, Max UPT, Max UPR, and Max UPE) that we proposed to solve this problem are able to earn significantly more utility than the comparison heuristics from the literature that do not consider utility and make permanent reservations instead of using place-holders (EASY Backfilling, Conservative Backfilling, and Multiple Queues). Because the system is oversubscribed and these comparison heuristics attempt to execute tasks in their FCFS arrival order, they will often run tasks that have had significant decay in their utility functions, resulting in less overall utility.

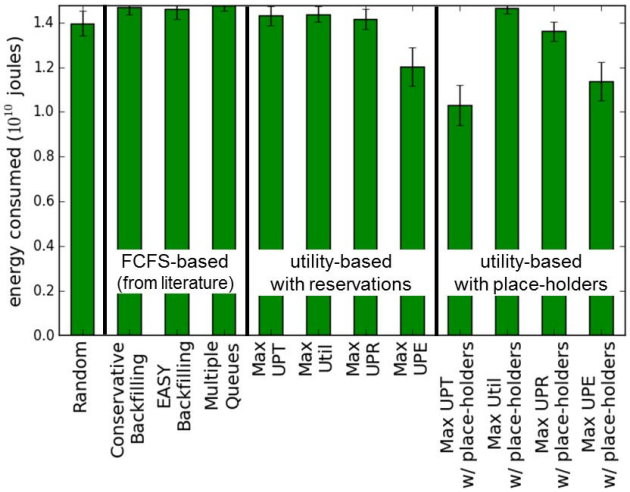
The energy-per-resource filter that we designed during this study outperformed the energy-per-task filter as seen in Figures 9(a) and 9(b) for a mean of 5,000 task arrivals. This is due to the increased ability of the energy-per-resource filter to execute tasks that have a higher amount of resources allocated to them (see (1)). Recall that tasks that have longer execution time in general have higher starting utility values (see Figure 6). The energy-per-task filter will almost always remove all options for these tasks because they use a large amount of resources, which increases the energy that they consume. We also examined how the energy-per-resource filter performs with a higher level of oversubscription created by a mean of 10,000 task arrivals per day. The results with this higher level of oversubscription can be seen in Figure 10. The relative performance of Max UPR with place-holders and Max UPE with place-holders is comparable to the results when a mean of 5,000 task arrivals per day was used, but the performance of the other place-holder heuristics has degraded such that Max UPR with place-holders and Max UPE with place-holders perform better with no overlapping 95% confidence intervals.

The performance of the comparison heuristics from the literature became significantly worse with this increase in oversubscription. With so many tasks arriving, it is more common for these heuristics to schedule tasks that earn insignificant amounts of utility. The permanent reservations for these tasks can extend far into the future preventing newly arriving tasks from running quickly. When compared with the results in Figure 9(a), the difference between the performance of the heuristics that earn the most utility (Max UPR with place-holders and Max UPE with place-holders) and the comparison heuristics from the literature such as Conservative Backfilling and EASY Backfilling has become more significant.

Even though the Max UPR with place-holders heuristic using the energy-per-resource filter is able to earn comparable utility to the Max UPE with place-holders heuristic for multiple levels of oversubscription, shown in Figures 9(a) and 10(a), the Max UPE with place-holders heuristic consumes less energy as seen in Figures 9(b) and 10(b). We consider Max UPE with place-holders to be the best heuristic that we have designed for use in energy constrained environments because it is able to earn utility comparable with all other high performing heuristics, while consuming less energy. We do not expect the overhead of our heuristics to be prohibitive when running on a typical computing environment's frontend or scheduling node. A single mapping event for the Max UPE with place-holders heuristic took 0.1 seconds on average. If the number of P-states was increased to 15, the Max UPE with place-holders heuristic still only takes 0.3 seconds to execute on average. In addition, with 15 P-states the performance of the heuristic in terms of utility earned and energy consumed is similar. A single mapping event for the Max UPR with place-holders and energy-per-resource filtering took less than 4 seconds on average. Both of these heuristics complete well within the scheduling interval of a typical cluster scheduler (usually approximately 60 seconds). In addition, all of our utility-aware heuristics outperform Random and FCFS-based techniques from the literature in terms of utility earned by the system over the course of a day.



(a) percentage of maximum utility for a mean of 10,000 task arrivals



(b) energy consumption for a mean of 10,000 task arrivals

Figure 10. Results for a mean of 10,000 tasks arriving per day. The utility-based heuristics utilize energy-per-resource filtering and task dropping with a dropping threshold of 0.5. In (a), a comparison of the percentage of maximum utility earned with 95% confidence intervals is shown. The energy consumption of each heuristic can be seen in (b) with 95% confidence intervals.

VI. Related Work

Many heuristics and techniques for resource management have been designed to operate in parallel dynamic HPC environments. Many of them, however, are designed for metrics that are not applicable to our oversubscribed, utility-based environment because they use fairness and time-based objectives as their performance measure (e.g., [12, 15, 17, 18, 25]). When designing resource managers for parallel resource allocation, it is common to start with Conservative Backfilling or EASY Backfilling and modify one of them to generate an improved heuristic. In [12], the authors designed an iterative Tabu search algorithm to improve the fairness of Conservative Backfilling. The Conservative Backfilling heuristic was also modified in [17] to create a heuristic that improves the average turnaround time of tasks. One of the reasons that our work differs from these is that we use the total utility earned over an interval of time as our performance measure.

Other authors have determined that utility functions are an effective metric for measuring the performance of resource managers in oversubscribed environments [20, 23]. This is done through surveying the literature in [20] and through the development of a framework for measuring supercomputer productivity in [23]. Our work extends these efforts by designing a resource manager that attempts to maximize utility earned while obeying an energy constraint. Monotonically decreasing functions, such as "value functions", also have been used to measure the performance of resource managers in various HPC environments and behave similarly to utility functions [8, 13, 14, 24]. Differences between these works and ours include that [8, 13, 24] do not consider heterogeneity and [24] does not consider parallel tasks.

The authors of [22] model a resource manager for a computing system where heterogeneous computing sites that are similar to our clusters are used, but they do not consider utility functions or energy consumption in their study. In addition, they measure the performance of their resource manager using utilization and average turnaround time. This is very different from our oversubscribed environment, which uses utility functions, total utility earned as a performance measure, and has an energy constraint.

Genetic algorithms are sometimes used to solve resource management problems because they are able to find very good solutions if they are given enough time to run. Utility was maximized using a genetic algorithm in [13], where the genetic algorithm was able to earn more utility than EASY Backfilling, Conservative Backfilling, and a Priority-FIFO heuristic. The drawback of genetic algorithms is that they require a significant amount of execution time to produce good results (e.g., the genetic algorithm in [13] had an average execution time of 8,900 seconds). When compared with our best heuristics, which take significantly less than a minute to execute on average, this long execution time is a major drawback to genetic algorithms. Being able to generate solutions to resource management problems quickly is very important in a dynamic environment. This is because nodes can be idle while the resource manager is making decisions and no work would be accomplished on those nodes during that time.

VII. Conclusions and Future Work

We designed and evaluated the performance of our utlity-aware resource allocation heuristics (Max Util, Max UPT, Max UPR, and Max UPE), and associated dropping and filtering techniques was measured in terms of the total system utility that was earned from the completion of parallel tasks in an oversubscribed HPC environment with an energy constraint. The concept of place-holders that we presented in [11] for a different environment, in addition to our new energy-per-resource filtering technique, allowed our utility-based heuristics to achieve significantly higher system utility than popular scheduling techniques from literature that do not consider utility functions and heterogeneity. Due to energy filtering, our Max UPR with place-holders heuristic was able to earn utility comparable to our energy aware Max UPE with place-holders heuristic, although Max UPE with place-holders is much more energy efficient.

A topic that we are interested in exploring in the future is the concept of preemption. We expect that having a resource manager that supports preemption will allow for improvement in the execution of critical tasks, in particular when there has been a period of low utility task arrivals that may fill up many of the nodes within an environment, which can cause high utility tasks arriving later to wait. This is especially important for an environment where tasks arrive dynamically (i.e., information about tasks that arrive in the future is not known). The most significant difficulty of designing and analyzing techniques and heuristics that utilize preemption is to limit the potential complexity of the problem. Similar to how it is not possible to explore all possible solutions of this scheduling problem in reasonable time, we cannot consider preempting every task individually to optimize the schedule. To limit this complexity, we will need to determine what type of task should be able to cause preemption among currently executing tasks, and will consider techniques for effectively selecting which tasks to preempt in a reasonable amount of time. Working with preemption may also require studying techniques for saving the state of a task so that the task can resume execution at a later point in time.

One advantage of our energy filters that would be interesting to explore is varying the energy leniency to control energy consumption at different times. For example, this could be used to limit energy consumption more aggressively at the times of day when electricity is most expensive.

Acknowledgments

The authors thank Bart Nielsen and Sarah Powers for their involvement on this project. We also thank the reviewers for their comments.

References

[1] S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, and S. Ali, "Representing task and machine heterogeneities for heterogeneous computing systems," *Tamkang Journal of Science and Engineering*, Special Tamkang University 50th Anniversary Issue, Vol. 3, No. 3, pp. 195-207, Nov. 2000. Invited.

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

[3] 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.

[4] A. B. Downey, *A Model For Speedup of Parallel Programs*, Technical Report UCB/CSD-97-933, EECS Department, University of California, Berkeley, 1997.

[5] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., 1979

[6] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation, "Device Power State Definitions," in *Advanced Configuration and Power Interface Specification*, Rev. 3.0, pp. 23, Sep. 2004.

[7] 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.

[8] E. Jensen, C. Locke, and H. Tokuda, "A time-driven scheduling model for real-time systems," in *Proceedings of the IEEE Real-Time Systems Symposium,* pp. 112-122, Dec. 1985

[9] 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*, Vol. 64, No. 8, pp. 2394-2407, Aug. 2015.

[10] 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.

[11] B. Khemka, D. Machovec, C. Blandin, H. J. Siegel, S. Hariri, A. Louri, C. Tunc, F. Fargo, and A. A. Maciejewski, "Resource management in heterogeneous parallel computing environments with soft and hard deadlines," *11th Metaheuristics International Conference (MIC 2015)*, 10 pp., June 2015.

[12] D. Klusaccek and H. Rudova, "Performance and fairness for users in parallel job scheduling," in W. Cirne, N. Desai, E. Frachtenberg, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, Vol. 7698 of *Lecture Notes in Computer Science,* pp.235-252. Springer Berlin Heidelberg, 2012.

[13] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems," *IEEE Transactions on Software Engineering*, Vol. 30, No. 9, pp. 613-629, Sep. 2004.

[14] C. B. Lee and A. E. Snavely, "Precise and realistic utility functions for user-centric performance analysis of schedulers," in *International Symposium on High Performance Distributed Computing (HPDC '07)*, pp. 107-116, 2007.

[15] D. A. Lifka. "*The ANL/IBM SP scheduling systems,*" In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Vol. 949 of *Lecture Notes in Computer Science*, pp. 295-303, Springer Berlin Heidelberg, 1995.

[16] 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. 107-131, Nov. 1999.

[17] A. Mishra, S. Mishra, and D. S. Kushwaha, "An improved backfilling algorithm: SJF-B," *International Journal on Recent Trends in Engineering & Technology*, Vol. 5, No. 1, pp.78-81, Mar. 2011.

[18] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12, No. 6, pp. 529-543, June 2001.

[19] M. A. Oxley, S. Pasricha, A. A. Maciejewski, H. J. Siegel, J. Apodaca, D. Young, L. D. Briceño, J. Smith, S. Bahirat, B. Khemka, A. Ramirez, and Y. Zou, "Makespan and Energy Robust Stochastic Static Resource Allocation of a Bag-of-Tasks to a Heterogeneous Computing System," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 26, No. 10, pp. 2791-2805, Oct. 2015.

[20] B. Ravindran, E. D. Jensen, and P. Li, "On recent advances in the time/utility function real-time scheduling and resource-management," in *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pp. 55-60, May 2005.

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

[22] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan, "Scheduling of parallel jobs in a heterogeneous multi-site environment," in D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, Vol. 2862 of *Lecture Notes in Computer Science*, pp. 87-104. Springer Berlin Heidelberg, 2003.

[23] M. Snir and D. A. Bader, "A framework for measuring supercomputer productivity," *International Journal of High Performance Computing Applications*, Vol. 18, No. 4, pp. 417-432, Nov. 2004.

[24] D. Vengerov, L. Mastroleon, D. Murphy, and N. Bambos, "Adaptive data-aware utility-based scheduling in resource-constrained systems," Technical Report 2007-164, Sun Microsystems, Inc., 2007.

[25] Y. Yuan, Y. Wu, W. Zheng, and K. Li, "Guarantee strict fairness and utilize prediction better in parallel job scheduling," *IEEE Transactions on Parallel and Distributed Systems*, Vol 25., No. 4, pp. 971-981, Apr. 2014.