Version of Record: https://www.sciencedirect.com/science/article/pii/S0167819117301862
Manuscript_a3ee4cc5d3e18c9fe1a88c8b272be8f4

Utility-Based Resource Management in an Oversubscribed Energy-Constrained Heterogeneous Environment Executing Parallel Applications

Dylan Machovec[a,*], Bhavesh Khemka[a], Nirmal Kumbhare[c], Sudeep Pasricha[a,b], Anthony A. Maciejewski[a], Howard Jay Siegel[a,b], Ali Akoglu[c], Gregory A. Koenig, Salim Hariri[c], Cihan Tunc[c], Michael Wright[d], Marcia Hilton[d], Rajendra Rambharos[d], Christopher Blandin[a], Farah Fargo[c], Ahmed Louri[e], Neena Imam[f]

[a] *Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO 80523, USA*
[b] *Department of Computer Science, Colorado State University, Fort Collins, CO 80523, USA*
[c] *Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ 85721, USA*
[d] *Department of Defense, Washington, DC 20001, USA*
[e] *Department of Electrical and Computer Engineering, George Washington University, Washington, DC, 20052, USA*
[f] *Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA*

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 a computing system is measured as the total utility earned by all completed tasks over some interval of time (e.g., 24 hours). We have designed, analyzed, and compared the performance of a set of heuristic techniques to maximize system performance when scheduling dynamically arriving parallel tasks onto a high performance computing (HPC) system that is oversubscribed and energy constrained. We consider six utility-aware heuristics and four existing heuristics for comparison. A new concept of temporary place-holders is compared with scheduling using permanent reservations. We

*Corresponding author
Email addresses: djmachov@rams.colostate.edu (Dylan Machovec), bhavesh.khemka@gmail.com (Bhavesh Khemka), nirmalk@email.arizona.edu (Nirmal Kumbhare), sudeep@colostate.edu (Sudeep Pasricha), aam@colostate.edu (Anthony A. Maciejewski), hj@colostate.edu (Howard Jay Siegel), akoglu@ece.arizona.edu (Ali Akoglu), koenig@acm.org (Gregory A. Koenig), hariri@ece.arizona.edu (Salim Hariri), cihantunc@email.arizona.edu (Cihan Tunc), michael.wright4@comcast.net (Michael Wright), mmskizig@verizon.net (Marcia Hilton), jendra.rambharos@gmail.com (Rajendra Rambharos), farahfarjo@email.arizona.edu (Farah Fargo), louri@gwu.edu (Ahmed Louri), imamn@ornl.gov (Neena Imam)

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 multiple energy-constrained oversubscribed HPC environments. We conduct an experiment with a subset of the heuristics on a physical testbed system for one scheduling scenario. We demonstrate that our proposed utility-aware resource management heuristics are able to significantly outperform existing techniques.

Keywords: heterogeneous computing, energy-aware computing, utility functions, resource management heuristics, parallel tasks, scheduling

---

1. 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 the time required to complete the computational work of the task. 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, e.g., the system cannot begin executing each task as soon as the task 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 to measure the performance of the system's 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 longer to complete the task (i.e., node types that are not as effective for that task) will still result in high system utilization, but provide delayed results for that task. Furthermore, because the system is oversubscribed, we would expect to always have near 100% utilization.

To effectively model the performance of an oversubscribed heterogeneous system, for this study we employ the concept of utility functions [1], which are appropriate for modeling the needs of DOE and DoD. Utility functions are monotonically-decreasing with time and represent the importance and urgency of a task. They define the utility earned by a task at the time of its completion. The performance of the overall computing 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.

Energy is an expensive and potentially limited resource required to operate HPC systems (e.g., [2, 3, 4, 5]). It has been found that attempting to scale

up current systems to achieve an exascale system would result in energy and power requirements that are currently not feasible. For example, the power requirement would be greater than a gigawatt [5]. In some environments, there is a limit on the amount of energy that is available in some interval of time [4, 6]. In this study, we constrained the amount of energy available to the HPC system each day. The general problem of mapping tasks onto a set of resources is known to be NP-hard [7]. 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. To effectively maximize system utility while satisfying this energy constraint, heuristics are needed. We also created a new energy filtering technique to improve the energy efficiency of our heuristics.

We designed four utility-aware resource allocation heuristics: Max Utility, Max Utility-per-Time, Max Utility-per-Resource, and Max Utility-per-Energy. We compared these to four approaches from the literature: Conservative Backfilling, EASY Backfilling, FCFS (first-come, first-served) with Multiple Queues, and Random [8, 9]. In addition, we designed two metaheuristics that switch between the Max Utility-per-Resource and Max Utility-per-Energy heuristics depending on how energy constrained the system is at the time of the task's mapping.

Many heuristics for the resource allocation of parallel tasks in HPC environments schedule using permanent reservations to allow for allocations of nodes to tasks in the future (e.g., [8, 9]). Because permanent reservations can be restrictive, we developed the concept of using temporary place-holders when scheduling. 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 include:

- the design of utility-aware heuristics and an energy-per-resource filtering technique with the goal of maximizing utility earned by parallel tasks while obeying an energy constraint in heterogeneous oversubscribed HPC environments;
- the design of new metaheuristics that make use of the strengths of different utility-based heuristics;
- the validation of the relative performance of the heuristics derived by the simulator through the use of an experiment on a physical testbed system for one scenario.

Preliminary versions of portions of this material appear in the 2015 Metaheuristics International Conference [10] and the 2016 Heterogeneity in Computing Workshop [11]. The differences between this work and the preliminary versions include: (a) the design, analysis, and evaluation of two new metaheuristics that in general result in improved performance; (b) the simulation of many more environments, which is used to further analyze the performance of the heuristics and the effect that different parameters have on the heuristics; and (c) experiments on a physical testbed are used to further evaluate the heuristics.

This paper is organized as follows. In Section 2, we define the HPC environment and problem we are addressing. Section 3 explains the resource management techniques that are utilized. The setup for our simulated environment is detailed in Section 4. The simulation analyses and comparisons are presented in Section 5. An experiment that we performed on a testbed system has its setup and results shown in Section 6. In Section 7, we discuss related work. Finally, in Section 8 we conclude and discuss future work.

2. Environment and Problem Description

2.1. 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 cores. The nodes that form each cluster are homogeneous, meaning that they are identical (and therefore have the same number and type of cores). The node architecture varies among clusters and each cluster can have different numbers of cores per node. We modeled cores 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 [12].

2.2. Workload and Environment Characteristics

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

Task types in this environment have estimated execution characteristics (execution time and energy consumed) that are deterministic and known to the system resource manager. We assume that this information is available through historical and experimental data. This assumption is a common practice in research for resource allocation (e.g., [13, 14, 15]). Tasks with similar execution characteristics belong to the same task type. Whenever a task arrives to the system 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 (tasks of the same type do not necessarily have the same 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 task types.

When a task is assigned to execute in the system, it is assigned to a set of nodes in one of the clusters. All nodes in this set will use the same P-state. Within each cluster, execution characteristics are defined by an Estimated Time to Compute (ETC) matrix and an Average Power Consumption (APC) matrix

[6]. The ETC matrix is used to specify the execution time of tasks for each task type, cluster type, 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 1. 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 some cases, a task type's execution time may increase (instead of decrease) with an increased number of nodes due to increased communication and synchronization overheads. In our simulations, if the number of nodes the task needs is not listed in the ETC matrix then its execution time is assumed to be between two values listed in the matrix, and we calculate its execution time using linear interpolation. We assume that all tasks require a number of cores between the minimum and maximum values provided in the ETC matrix.

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 2, 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 utility function of the task, to map the tasks to nodes in one of the clusters.

2.3. Utility Functions

Our performance metric is based on utility, a flexible measure of the importance of a task. Utility functions [1] are monotonically decreasing functions that define the utility that a task earns upon completion, and depend on the amount of time that has passed since the task was submitted to the system as depicted in Figure 3. 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. Each task has an associated utility function that may differ from the utility functions of other tasks.

2.4. Problem Statement

We defined the system utility earned over a day as the sum of utility earned by all tasks that are completed by the system during that day. This also includes a portion of the utility earned by each task if the task would be partially completed during that day. This can occur when the task has an execution time greater than the amount of time remaining in the day when its execution begins and it has not yet finished its execution when the day reaches its end. For example, if a task were to complete 70% of its total execution time during

day i and 30% of its total execution time during day $i+1$, then the utility earned for this task during day i would be 70% of the task's final utility and the utility earned during day $i+1$ 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 each day. *The goal of our resource manager was to maximize the utility earned by the system subject to the energy constraint of the system.*

3. Resource Management

3.1. Mapping Events

Mapping is the process of assigning and scheduling tasks to the nodes of the HPC system. When a task arrives to 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 to assist in maximizing system utility. First, some of the tasks are dropped to tolerate oversubscription (described in Subsection 3.3). Next, energy filtering (detailed in Subsection 3.8) attempts to improve energy efficiency by limiting the allocation options for tasks. Finally, one of the heuristics defined in Subsections 3.4, 3.5, or 3.6 is used to make the final resource management decisions. In the environment we considered, 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.

3.2. Permanent Reservations and Place-holders

A permanent reservation marks the resources that will be allocated to a task at some point in the future. The number of *cores allocated* to a task is equal to the number of nodes that are allocated to the task multiplied by the total number of cores on each node (even if only a subset of the cores in a node are used by the task). 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:

$$resources\ allocated = execution\ time \times cores\ allocated. \tag{1}$$

If a task cannot begin executing immediately on available nodes, a permanent reservation 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. 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. We created place-holders as an alternative to permanent reservations. A place-holder 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 to begin execution sooner if those tasks

would earn more utility than the tasks originally scheduled to those resources during the last mapping event.

3.3. 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 allows the shortest execution time for the task. This calculated value 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. Determining the optimal dropping threshold may be accomplished through simulations.

3.4. Comparison Heuristics

3.4.1. Overview

Four of the heuristics we considered were for the purpose of comparison to our utility-aware resource management techniques. Three of these heuristics are commonly used in parallel scheduling. In addition, we consider a Random heuristic as an additional point of comparison. The process used by these heuristics to select a subset of nodes within a cluster is described in Subsection 3.7

3.4.2. 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.

3.4.3. Conservative Backfilling

This heuristic, described in [9], 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 0 as the P-state, indicating the highest performance for a given node.

3.4.4. EASY Backfilling

Extensible Argonne Scheduling sYstem (EASY) Backfilling, from [8], 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 uses P-state 0.

3.4.5. FCFS with Multiple Queues

The FCFS (first come, first served) with multiple queues heuristic, designed to model systems such as CSU's ISTeC Cray [16], 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 (based on Equation (1)) that any task will be allocated, averaged over the clusters. This average amount of resources is then used 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 specific lower threshold, upper threshold, and the number of tasks assigned from each queue in each iteration are examples of what may be used in a system. Implementations of this heuristic on other systems may use different values for these parameters. The motivation for this heuristic is to attempt to balance the tasks being assigned to the system based on resources needed.

3.5. Utility-Aware Heuristics

3.5.1. 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 [17], which has been used successfully in many environments (e.g., [18, 19]), but has not been explored in an oversubscribed energy constrained environment with parallel tasks. All of these utility-aware heuristics have a similar structure that defines their execution, but each utilizes a different objective measure.

3.5.2. Heuristic Objective Measures

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

$$Util = value\ of\ the\ task's\ utility\ function\ at\ completion, \tag{2}$$

$$UPT = Util/the\ task's\ execution\ time, \tag{3}$$

$$UPR = Util/resources\ allocated\ to\ the\ task, \tag{4}$$

$$UPE = Util/energy\ consumed\ by\ the\ task. \tag{5}$$

We defined four heuristics, Max Util, Max UPT, Max UPR, and Max UPE using the objective measures listed in Equations (2)-(5), respectively. Max Util, Max UPT, and Max UPE were used in our work in [1, 6] with an energy constraint, but only for serial tasks. Thus, the work presented in this study is significantly different because the heuristics are designed for parallel tasks that are assigned to sets of nodes.

3.5.3. 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 4.

3.5.4. 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 a cluster, 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 5 to 11. This algorithm also can 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 the set of 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 do
2:      for each task in the set of mappable tasks do
3:          find nodes/cluster/P-state combination
            that maximizes UPE for the task
4:      end for
5:      select task from the set of mappable tasks with nodes/cluster/P-state
        combination that has the highest maximum UPE
6:      if selected task can start execution immediately
        with that nodes/cluster/P-state combination then
7:          assign selected task to that nodes/cluster/P-state combination
8:      else
9:          create a permanent reservation for selected task
            on that nodes/cluster/P-state combination
10:     end if
11:     remove task from the set of mappable tasks
12: end while
```

3.6. Metaheuristics

3.6.1. Overview

In some cases, none of the heuristics described are well suited to a particular environment. In this situation, a strategy based on a concept in [20], permits switching between heuristics depending on the current state of the system. We designed two metaheuristics to achieve good performance regardless of the energy constraint. These metaheuristics switch between Max UPE and Max UPR depending on conditions defined below.

3.6.2. Event-Based Metaheuristic

The Event-Based metaheuristic chooses one of Max UPE and Max UPR at the start of each mapping event and uses that heuristic for the entire mapping event. At any given time during the day, we define a "goal energy," which is the energy that should be consumed up to a specific point of the day. This goal energy could also be determined using known task arrival data to potentially improve the metaheuristic, but because the environment in this study has an unknown dynamic task arrival pattern, we consider the case where the goal is to consume energy at a constant rate so that tasks arriving at any point during the day will have energy to use. At the start of each mapping event, we calculate the sum of energy consumed since the beginning of the day and the energy that

will be consumed if all tasks in the current mapping also execute (i.e., currently executing tasks and tasks with reservations or place-holders). If this energy is greater than the goal energy, the Event-Based metaheuristic will use Max UPE because the system has been consuming energy at a rate above the goal. Otherwise, the metaheuristic will select Max UPR.

3.6.3. Task-Based Metaheuristic

The Task-Based metaheuristic will initially select Max UPE or Max UPR using the same strategy as the Event-Based metaheuristic described above. If the metaheuristic chooses to use Max UPR, it will use the heuristic to assign tasks one by one. As tasks are assigned, the sum of energy consumed since the beginning of the day and the energy that will be consumed if all tasks in the current mapping also execute is updated. Once this sum reaches the goal energy, the heuristic will switch to Max UPE and will finish the mapping event by assigning tasks with Max UPE.

3.7. Finding Allocation Options for a Task

We designed a technique to select a node allocation for a task within a cluster (line 5 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., Max UPE) within each cluster is achieved when that task finishes execution as soon as possible in that cluster with that P-state. The execution time of a task 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 earliest possible start time for the task. The allocation options that are considered for a task are the earliest possible start time for each P-state/cluster combination. This strategy was used for all of the heuristics that we present in this paper (i.e., the comparison heuristics discussed in Subsection 3.4, the utility-aware heuristics detailed in Subsection 3.5, and the metaheuristics described in Subsection 3.6).

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, which is only applied if there is a tie in the first 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 4a, where a task t is requesting three nodes. The earliest time when three nodes are available is a time when four nodes (*n3*, *n4*, *n5*, and *n6*) 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 red arrows in Figure 4 show the size of the idle voids being compared in this step). 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.

3.8. Energy Filtering

3.8.1. 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, which could result in lost utility due to the inability to execute potentially high utility tasks that arrive after the energy constraint has been reached. This is just a heuristic approach and its effectiveness will need to be evaluated to determine what is suitable for a typical expected environment.

3.8.2. Energy-per-Task Filtering

We calculate the energy-per-task budget for a task as the fair share of energy that the task is permitted to consume. This budget, extended from our serial version of this energy filter in [6] 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:

$$\begin{aligned} &resources\ remaining = \\ &\quad \sum_i unallocated\ time\ remaining\ on\ node\ i \times cores\ on\ node\ i, \end{aligned} \tag{6}$$

$$\begin{aligned} &estimated\ number\ of\ tasks\ remaining = \\ &\quad min(\frac{resources\ remaining}{average\ resources\ used}, \frac{energy\ remaining}{average\ energy\ consumed}), \end{aligned} \tag{7}$$

$$\begin{aligned} &energy\text{-}per\text{-}task\ budget = \\ &\quad leniency\ factor \times \frac{energy\ remaining}{estimated\ number\ of\ tasks\ remaining}. \end{aligned} \tag{8}$$

A leniency factor is included that can be used to adjust the filter. As the leniency factor is increased, the filter will allow more options for each task. In our simulations, we set this factor by performing a parameter sweep to find the best possible leniency value. Any task allocation options where the task would consume energy greater than the energy budget are not considered by heuristics.

3.8.3. Energy-per-Resource Filtering

In our previous work, there were only serial tasks resulting in 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:

$$\textit{energy-per-resource budget} = \textit{leniency factor} \times \frac{\textit{energy remaining}}{\textit{resources remaining}}. \quad (9)$$

Again, we multiply by a leniency factor determined by simulations 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 Equation (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.

4. Simulation Setup

4.1. Overview

The simulation setup described in this section was designed based on discussions with researchers from ORNL and DoD. We generated 48 simulation trials as described in Subsection 4.2. We simulated a total of 28 hours, but only analyzed 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.

4.2. Generation of Compute System and a Synthetic Workload

4.2.1. Compute System

The compute system we simulated 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. Special-purpose clusters are clusters with specialized hardware that are designed to execute only specific tasks. For example, the nodes of a special-purpose cluster may have GPUs available and would only execute tasks that can utilize the GPUs. 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.

4.2.2. Workload

In our simulations, two workloads were considered. The first has a mean of 5,000 parallel tasks arriving per day and the other has a mean of 10,000 parallel tasks arriving per day. Each of the tasks belongs to one of 100 task types that we generate for each simulation trial with different ETC values and APC values. Of these 100 task types, 60 are tasks that can execute on general-purpose clusters

and 40 are tasks that execute on special-purpose clusters. General-purpose tasks can only run on the general-purpose clusters. Of the tasks that execute on special-purpose clusters, 20 types can execute only on one of the special-purpose clusters and the other 20 types can only execute on the other special-purpose cluster. Because they cannot execute on the same clusters, the only interaction between the tasks that execute on general-purpose clusters and the tasks that execute on special-purpose clusters is the shared system energy constraint.

4.2.3. Utility Functions

To describe a task's utility function, we used three parameters: priority, urgency, and utility class [1]. The priority (or starting utility) and urgency parameter for the utility function of each task type was generated using the distribution of priority and urgency shown in Table 2 (from [1]). The actual starting utility value was chosen uniformly from the starting utility range associated with each priority level. For each task of a task type, a utility class (defined in Subsection 2.3) is randomly selected from one of 20 that we generated for our simulation studies in [1].

4.2.4. Single Core Execution Time

The execution time for each task type on a single core for one of the clusters is sampled from a Gaussian distribution. Because we assumed there is a correlation between the single core execution time of a task and the starting utility value, the mean of the Gaussian distribution is selected based on the starting utility of the task type. This was because, in our intended environment, longer running tasks are generally of higher importance. The perfectly correlated values for starting utility and single core execution time were defined such that task types with the minimum possible single core execution time (set to 1 hour) had the minimum possible starting utility (set to 1). Similarly, task types with the maximum possible single core execution time (set to 18 hours) had the maximum possible starting utility (set to 8). The perfectly correlated values are obtained through linear interpolation using these perfectly correlated end points. The perfectly correlated values were used as the mean values for the Gaussian distribution described above for determining the single core execution time for each task type. 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 5. This correlation was 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) was modeled using the COV method from [21] with a COV parameter of 0.3. To generate this heterogeneity using the COV method, the execution time on each other cluster is sampled from a gamma distribution with the COV of 0.3 and a mean equal to the correlated single core execution time described above. The entries of the APC matrix were generated using the COV method for generating ETC matrices [21]. The power consumption on one of the clusters is generated by sampling a gamma distribution with a mean power consumption of 133 watts and a COV of 0.2. The power consumption for each of the other clusters is then

sampled from a new gamma distribution with a COV of 0.2 and a mean equal to the power consumption obtained for the first cluster.

4.2.5. Individual Task Arrivals

Once we had the completed set of task types, individual tasks were generated for each task type. We defined 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 obtained from Table 3 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 generated the arrival pattern for a task type. If the task type requires fewer 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.

4.2.6. Parallel Execution Time Scaling

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 [22]. We use the high variance model, which is defined in terms of two parameters A (the average degree of parallelism for the program) and σ (the COV of the parallelism for the program). The σ value used in our simulations is sampled uniformly between 4 and 10 for each task type and A is equal to the average number of nodes requested by the task type. This model represents tasks that have a sequential component of length σ and a parallel component with an execution time of 1 when the task is given its maximum parallelism. The parallel component has a maximum parallelism of $A + A\sigma - \sigma$. The execution time of a task in terms of the number of nodes, n, allocated to it is then defined as:

$$T(n) = \begin{cases} \sigma + \frac{A + A\sigma - \sigma}{n} & 1 \leq n \leq A + A\sigma - \sigma \\ \sigma + 1 & n > A + A\sigma - \sigma \end{cases}. \tag{10}$$

4.2.7. P-states

Cores may have many P-states. For our simulation study, we assumed they each had three P-states. This provides for a choice between the lowest P-state, an intermediate P-state, and the highest P-state. We also ran simulations with fifteen P-states and found that the relative performance of the heuristics was the same as using three P-states. Because of this, we consider three P-states to be a good sample for modeling the advantages that can be gained from

having multiple P-state options. 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 were defined using a "power scaling factor" for each P-state. This factor was used to scale the average power usage and execution time of each task for that P-state. The three P-states have power scaling factors of 1.0, 0.75, and 0.5. A "randomness factor" also was used so that the power scaling factor is not the same for all combinations of task types and clusters. Each randomness factor was 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 power consumption scaling for each of the three P-states is determined by sampling from one of three gamma distributions (each P-state has a different distribution). These gamma distributions have means equal to the power scaling factor associated with that P-state multiplied by a randomness factor (generated as described above). In addition, the gamma distributions have a COV of either 0.03 for general-purpose tasks or 0.02 for special-purpose tasks. The execution time scaling for each P-state was determined in a similar way to the power scaling. The execution time scaling for each of the three P-states is determined by sampling from one of three gamma distributions (each P-state has a different distribution). These gamma distributions have means equal to the square root of the product of the power scaling factor associated with that P-state and a randomness factor [6]. The final execution time scaling was found by taking the reciprocal of this value so that the execution time of the task type is increased when there is less power.

4.3. Generating a Workload from a Real System Trace

We also simulated a system that used a workload of tasks generated from the log of the Curie Supercomputer in France from Dror Feitelson's Parallel Workloads Archive [23, 24]. We used the "clean" version of the Curie trace and selected 48 days from the trace for our simulations. Because we simulate 28 hours in total, we use the last four hours from the previously simulated day as the first four hours of the 28 hours. In addition, we removed any tasks that requested more than 4,096 cores from the trace to keep the size of the simulations tractable. Over the 48 days of log data, this resulted in the removal of 1,114 tasks out of 92,298 tasks in total (1.2% of the tasks).

From this trace, we took each task's arrival time, execution time, and the number of cores that were allocated to it. Unlike the environment we considered in this study, the data we took from the Curie trace was for a homogeneous system. To generate a workload for a heterogeneous system, we use the execution time as the task's execution time on one of the clusters of the simulated system and generate values for the other clusters using the method described above in Subsection 4.2. In addition, the size of the simulated system in cores is equal to a fraction of the 92,160 cores of the Curie system. This fraction is varied between 10% and 80% of the cores in our simulations. We always use a fraction of the cores to ensure that the system is oversubscribed (all tasks in the Curie

trace started and finished execution on the real Curie system). All other aspects of the workload and system are generated using the same methods described above in Subsection 4.2.

4.4. Resource Management Parameters

4.4.1. Dropping Threshold

The dropping threshold for our resource manager was set to 0.5 for the majority of our simulations. We also simulated scenarios where there was no dropping or a dropping threshold of 0, 0.1, or 0.3. Using a dropping threshold of 0.5 means that tasks that could no longer earn utility greater than 0.5 if they were to start execution immediately in their fastest cluster were 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). Lower dropping thresholds resulted in all heuristics earning less or equal utility than they did with a dropping threshold of 0.5. In actual practice, the threshold can be set based on simulations modeling the real system environment to be used. Dropping may also be disabled entirely, but this could greatly decrease system performance in terms of utility earned depending on which heuristic is used.

4.4.2. 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 2.3). The leniency factors for the two energy filters were both selected empirically using simulations, by varying the energy leniency factor for the Max UPR heuristic with place-holders as seen in Figures 6 and 7 for a mean of 5,000 tasks arriving per day. The 95% mean 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 energy leniency factors for a mean of 10,000 task arrivals were determined using the same method. In practice, the leniency factors can be set based on the results of simulations modeling the real system environment to be used.

4.4.3. 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 energy-aware approaches.

The energy constraint for our simulations for a mean of 5,000 tasks arriving was initially 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 gigajoules, which was used for most of our simulations. This provided a good starting point to ensure that the system would be constrained in terms of energy. In addition, we varied the energy constraint for this system from 8 gigajoules to 18 gigajoules to study a wider range of constraints. For our study of the workload generated using the Curie trace, we varied the energy constraint from 8 gigajoules to 26 gigajoules.

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 gigajoules. In a real system, the energy constraint would be set by the system administrator.

5. Simulation Results

5.1. Comparing 5,000 and 10,000 Tasks per Day

In Figure 8a, 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 8b with an energy constraint of 12 gigajoules. Results using the energy filters are not shown for the UPE heuristic because they had identical performance in this environment. Using either energy filtering technique allows the other utility-based heuristics 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 8 are based on the 48 simulation trials. The set of comparison heuristics (Random, Conservative Backfilling, EASY Backfilling, and Multiple Queues) did not use the energy filters because they are not applied to these heuristics in the literature.

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 comparison heuristics obtain close to 35% of the maximum system utility on average, while the worst performing utility-aware heuristics earn an average of 55% of the maximum system utility. Finally, the best performing heuristics have an average utility of around 73% of the maximum system utility.

The energy-per-resource filter that we designed during this study outperformed the energy-per-task filter, earning 8% more utility on average in the case of Max UPR with place-holders, as seen in Figures 8a and 8b 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 Equation (1)). Recall that tasks that have longer execution time in general have higher starting utility values (see Figure 5). The energy-per-task filter will almost always remove all options for these tasks because of the large amount of energy that they consume due to the increased amount of resources allocated to them (Equation (1)). 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 9. The relative performance of Max UPR with place-holders and Max UPE with place-holders is comparable to the results for a mean of 5,000 task arrivals per day, but the performance of Max UPT with place-holders has degraded such that the other place-holder heuristics perform better with no overlapping 95% confidence intervals, obtaining up to 50% more utility in the case of Max UPR with place-holders. Max UPT does poorly because it prioritizes tasks tasks with shorter execution times (it does not consider the number of nodes a task is assigned to). Tasks that are parallelized over many nodes will often have the shortest execution times. Max UPT will prioritize these tasks, which is inefficient in terms of resources. This results in especially poor performance because of the high level of oversubscription in this environment.

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 Figure 9a is compared with the results in Figure 8a, the difference between the performance of the heuristics that earn the most utility (Max UPR with place-holders and Max UPE with place-holders, which have 49.5% and 48.7% of the maximum utility on average, respectively) and the comparison heuristics from the literature such as Conservative Backfilling and EASY Backfilling, which obtain averages of only 15.3% and 7.1% of the maximum utility, respectively, 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 8a and 9a, the Max UPE with place-holders heuristic consumes less energy as seen in Figures 8b and 9b, where the average energy consumed by Max UPR with placeholders increases by 20% when compared to Max UPE with place-holders. 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 when simulating the results shown in Figure 8. If the number of P-states were increased to 15, our simulations showed that the Max UPE with place-holders heuristic would only take 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. When considering the larger number of tasks shown in Figure 9, the Max UPE with place-holders heuristic took less than two seconds to execute. A single mapping event for the Max UPR with place-holders and energy-per-resource filtering took three seconds on average when simulating the results shown in Figure 8 and the Max UPR with place-holders heuristic with energy-per-resource filtering took 80 seconds in the larger simulations shown in Figure 9. With the smaller number of tasks, both of these heuristics complete well within the scheduling interval of a typical cluster scheduler (usually approximately 60 seconds). Max UPE with place-holders continues to execute well within this scheduling interval even for larger numbers of tasks.

5.2. Evaluation of the Metaheuristics

We also designed two metaheuristics that combine Max UPR and Max UPE. Results comparing these heuristics to the other heuristics are shown in Figure 10. In these results, all heuristics utilize a dropping threshold of 0.5 and results are not shown for the utility-based heuristics with permanent reservations because permanent reservations never performed better than place-holders in any of our simulations. These results show that the metaheuristics are able to use the entire energy budget for the day while earning an average utility that is comparable to Max UPE. The advantage of the metaheuristics over Max UPE is discussed in Subsection 5.4 Because the Event-Based metaheuristic requires simpler operations, it runs faster than the Task-Based Metaheuristic, but their performance in terms of utility earned is the same, therefore we consider the Event-Based metaheuristic to be a better metaheuristic. Another significant advantage to the metaheuristics over Max UPR is that they perform well even without the use of an energy filter. In our simulations, using an energy filter resulted in a significant increase in the execution time of the heuristics. Because the metaheuristics do not use an energy filter, the overhead of running these heuristics is significantly lower than Max UPR with place-holders and the energy-per-resource filter. The Event-Based Metaheuristic completes mapping events in the simulations used to generate Figure 10 with an average execution time of 0.1 seconds compared to an average execution time of three seconds for Max UPR with place-holders and the energy-per-resource filter.

5.3. The Effects of Varying the Dropping Threshold

The dropping threshold was varied for the 5,000 task environment to obtain the results shown in Figure 11. The figure shows the percentage of maximum utility earned by all of the heuristics for each dropping threshold. In this environment, the energy constraint is 12 gigajoules. These results demonstrate

the effect of various dropping thresholds below the minimum starting utility of any task in the system. For the utility-based heuristics, dropping does not have a significant effect on performance. For the comparison heuristics other than Random, there is a significant increase in performance when the dropping of tasks is enabled and the dropping threshold is greater than 0. For example, when increasing dropping threshold from 0 to 0.1, Conservative Backfilling, EASY Backfilling, and Multiple Queues see increases of 28%, 20%, and 28% in their average utility earned, respectively. This is because these heuristics do not normally consider utility and consider the tasks in FCFS order. This means that the oldest tasks, which are the most likely to have a large decay in their utility functions, will be scheduled first. Using dropping with these heuristics ensures that all tasks that get scheduled by the heuristic will earn some utility. This effect is not as significant for the Random heuristic (there is an increase of 5% when increasing the dropping threshold from 0 to 0.1) because it often skips over some of the tasks that would not earn a significant amount of utility.

5.4. Energy Constraint Analysis

In Figure 12a, the percentage of maximum system utility earned in eleven energy constrained environments for a mean of 5,000 tasks is shown. Here, all heuristics employed task dropping. The energy consumption for these results can be seen in Figure 12b. The confidence intervals in Figure 12 are based on the 48 simulation trials.

It can be seen in Figure 12a that for environments that have a tight energy constraint, the Max UPE heuristic and both metaheuristics (Event-Based and Task-Based) are able to earn the most utility. On average, they earn 60% of the maximum system utility in the case with an 8 gigajoule energy constraint. The reason for this is because these heuristics consider the energy consumption of tasks when making mapping decisions. Max UPE will always choose the most energy efficient option for any task while the metaheuristics will attempt to use energy at a constant rate throughout the day so that tasks arriving later in the day have the opportunity to execute.

In the environments with a loose energy constraint, the Max Util heuristic, Max UPR heuristic, and the metaheuristics are able to earn the most utility. In the case with an 18 gigajoule energy constraint, they earn almost 80% of the maximum system utility on average. This is because in these environments energy is not a significant constraint and it is more important to use the system resources efficiently by selecting the tasks that would earn the highest utility. In addition, the metaheuristics behaved similarly to Max UPR because with a loose energy constraint they rarely select Max UPE. The utility-aware heuristics that do not consider energy also see a very significant drop-off in terms of utility earned as the energy constraint becomes tighter.

For the cases with intermediate energy constraints, the metaheuristics earn the highest utility. In the environment with a 15 gigajoule energy constraint, the metaheuristics get 79% of the maximum system utility while the Max UPE and Max UPR heuristics obtain 74% of the maximum system utility on average and the corresponding 95% confidence intervals do not overlap. This is because

they are energy-aware, but do not always choose the most energy efficient option for a task if a task with higher utility is available. In these environments, Max UPE will maximize its energy efficiency, resulting in reduced utility earned due to not using the entire energy constraint.

5.5. Impact of Integrating an Energy Filter

Results for the same set of environments using the energy-per-resource filter are shown in Figure 13. Figure 13a shows the percentage of maximum system utility earned by each of the heuristics with this filter. This filter improves the utility earned by the utility-based heuristics when their energy consumption without an energy constraint is greater than the energy constraint. In all cases, the Event-Based metaheuristic earns utility that is comparable to the other best performing heuristics. The leniency factor used in these simulations was the one found in Section 4 for the case with a 12 gigajoule energy constraint. Figure 13b shows the energy consumption for each of the heuristics. These results suggest that it is only worth spending time determining the best leniency factor for very tight energy constraints because the metaheuristics are able to earn the highest utility for the more the forgiving constraints without energy filtering.

5.6. Analyses with Curie Workload Arrival Trace

5.6.1. Results without an Energy Constraint

Figure 14 shows the percentage of maximum system utility earned in 15 environments using the workload generated from the Curie system trace. The different cases shown for each heuristic represent different system sizes. The size of each system is given as a percentage of the size in cores of the actual Curie system associated with the trace. Here, all heuristics made use of task dropping. The energy consumption for these results can be seen in Figure 14b. The confidence intervals in Figure 14 are based on the 48 simulation trials.

Figure 14a shows that the utility earned by the metaheuristics, Max UPT, and Max UPR is the highest among the heuristics. This makes sense because this environment is not energy constrained and the heuristics that do not consider the energy consumed by a task perform the best. The reason for the poor performance of Max Util is likely because there are periods in this trace where very large numbers of small tasks arrive to the system needing to be mapped. Because the maximum utility given to a large task is eight, if it is possible to execute more than eight small tasks that would earn one utility in that time then the larger task should not be executed. Max Util underperforms in comparison to the other heuristics for these task arrival cases because it only considers maximizing the utility earned by each task. The other utility aware heuristics are able to account for this (by considering time or resources) and select the smaller tasks instead. Max UPE is able to achieve high performance in these scenarios because tasks with a very short execution time use less energy than the tasks with a long execution time.

5.6.2. Results with an Energy Constraint

Figure 15 shows the performance in the Curie trace environment for a simulated system with 80% of the total number of cores of the actual Curie system. These results are shown for a range of energy constraints. The 95% mean confidence intervals in these results are larger than in the other results because variance between task arrival patterns on different days of the trace can significantly affect the performance of the heuristics. The heuristic that earns the highest average utility for the tighter energy constraints, as seen in Figure 15a, is Max UPE. This is because the Max UPE heuristic always chooses the most energy efficient mapping option for each task. The metaheuristics still perform well relative to every utility-aware heuristic except for Max UPE. However, they do not perform as well in some of these environments because they attempt to keep the rate of energy consumption constant throughout the day. In this environment, there are often very large bursts of tasks that arrive all at once. This means that the rate of energy consumption throughout the day should not be assumed to be constant and energy should be saved for the periods when a large set of tasks is arriving. The arrival pattern of the tasks is not known in advance because the environment is dynamic. If the arrival pattern was known, then the metaheuristics could be modified to consume energy at a rate consistent with the rate of arriving tasks in the system, which may result in increased performance. This arrival pattern could also be approximated through the use of historical data. The other utility-aware heuristics do not perform as well as Max UPE or the metaheuristics, but still perform better than the comparison heuristics (Random, Conservative Backfilling, EASY Backfilling, and FCFS with Multiple Queues). Similar to the results with a fully synthetic workload, the energy consumption (shown in Figure 15b) is lowest for the Max UPE heuristic. The other heuristics have energy consumption that stays closer to the energy constraint.

5.7. Discussion of Results

The results shown in this section indicate that: (a) the use of place-holders results in more utility earned than permanent reservations; (b) utility-based heuristics earn more utility than the comparison heuristics; (c) in most environments, the Event-Based Metaheuristic and Task-Based Metaheuristic earn the highest utility; (d) the energy filtering techniques are only beneficial when the energy constraint is very tight; and (e) when the Max UPE heuristic earns utility equal to the other best performing heuristics it often consumes significantly less energy. Based on these results, it is clear that the best heuristics for these HPC environments are Max UPE and the Event-Based Metaheuristic. In addition, it can be beneficial to add the energy-per-resource filter (which was shown to be better than the energy-per-task filter) to these heuristics if the energy constraint is tight, as shown in the difference between Figures 12 and 13. We also evaluated and compared several of our heuristics on a testbed system as discussed in the next section.

6. Experiment

6.1. Experimental Setup

We conducted an experiment on a testbed system, where we implemented several of our heuristics. We used an IBM HS22 blade server with four homogeneous clusters. Each cluster consists of two Intel Xeon X5650 six core processors (2.67 GHz), with 24 threads (two threads per core) and 24GB RAM (memory). Each cluster runs the Kernel-based Virtual Machine (KVM) hypervisor [25]. We treated each of the 24 threads as an individual CPU for a virtual machine (VM). A VM in the experimental setup corresponds to a node in the simulation environment. For consistent use of terminology in this paper, we referred to VM as a node in the experimental study. We recorded the power consumption of each node using the IBM Advanced Management Module [26]. The nodes in a given cluster have the same pre-determined configuration (core count and RAM allocation). However, the node configuration across the clusters shows heterogeneity as illustrated in Table 4 (columns 2-4). The allocation of cores and RAM varies across clusters to emulate a heterogeneous environment.

We used a subset of NAS-NPB MPI benchmarks [27] for the experimental evaluations. This subset included an integer sort (IS), Poisson equation solver (MG), and conjugate gradient (CG). Similar to the simulation study, we assumed that each benchmark had a fixed number of required cores, e.g., the IS.8 is an eight thread task with the requirement of eight cores.

Each node on cluster 4 has more cores than the nodes of the other clusters. This means that applications scheduled on cluster 4 will often have less inter-node communication than the other clusters. For example, a task that requires eight cores will be able to run entirely on one node in cluster 4 and would have no inter-node communication. On the other clusters, it would need to use additional nodes and may have to communicate between them. Because of this, if the memory requirement (i.e., the minimum amount of memory required to execute the benchmark without significant slowdown due to swapping) is met for a given benchmark by all the clusters, then cluster 4 will always have better execution time than the others. In the simulation study, the purpose behind the heterogeneity across the clusters was to have best execution time for different task type on different clusters. For our experimental study, where we emulate heterogeneity, this is only feasible if certain benchmarks are selected such that their memory requirements may not be satisfied by all the clusters but only a few. IS_CG.8 is an eight core customized task created by combining IS and CG benchmarks. The memory requirements for IS and CG are 2GB and 1GB, respectively. The combined memory requirement for the IS_CG is 3GB. This task has minimum execution time on cluster 3, because 3GB memory is available for eight cores (two nodes). Cluster 4 provides only 2GB memory for eight cores (one node), which leads to accessing of swap memory and increase in the execution time. Table 5 shows the estimated time to compute (ETC) matrix for our benchmarks. MG.8 has a minimum memory requirement of 3.5 GB. Therefore, execution of MG.8 on clusters 3 and 4 is not feasible due to unavailability of minimum required memory with eight cores.

Even after addressing the heterogeneity issue for clusters and benchmark execution time, we still observed that the maximum power consumption was approximately same for a given benchmark on all the clusters. Therefore, the dynamic energy consumption for a benchmark became linearly proportional to its execution time and preference of resource allocation for this benchmark was same for the UPR and UPE heuristics. This created a challenge for differentiating the UPE from UPR. To overcome this challenge, we set the maximum limit on P-states (operating frequency) and maximum possible C-states (sleep states) for each cluster as shown in the last two columns of Table 4. By setting a limit to the maximum P-state, we were able to vary the maximum dynamic power consumption for a given benchmark across the clusters. With lower range on C-states, we were able to increase the idle power consumption on clusters 1 and 3, and this gave us more flexibility in varying the dynamic power consumption for each task across all the clusters. Table 6 represents the estimated energy consumption to compute (EEC) for all the benchmarks on different clusters. In our experimental study, each benchmark represents a task.

6.2. Workload Generation, Experimental Flow, and Data Collection

Our synthetic workload was designed such that at a regular interval of 120 seconds (on average) a new task arrived with a probability of 40% for the task being IS.8, 40% for the task being IS_CG.8, 10% for the task being MG.16, and 10% for task being MG.8. The task inter-arrival time was selected as 120 seconds (experimentally determined) so that the system could achieve non zero utility for more than 98% of the tasks. IS.8 and IS_CG.8 were given a higher probability because they required fewer core count than MG.16 and provided more options for cluster selection compared to MG.8. We generated a workload based on these constraints for a duration of three hours. Each task was associated with a utility function. For each task, maximum utility value was randomly selected in the integer range of one to four with uniform probability. For each task the maximum utility was set to decay after 2.5 times of task's minimum execution time. The utility function was set to reach 0 at four times of task's minimum execution time.

Our experimental setup consisted of a centralized scheduler (CS) and four local schedulers (LS) (one per cluster). We used the CS for running heuristics on the input workload, and tracking the total utility earned by the scheduler and the energy consumption of the whole system. We used the LS to collect the list of tasks from the CS, schedule them on the available nodes, monitor their progress, and collect cluster power consumption. We used the internal clock of the CS to compare the time stamp of each task in the workload. Inside CS, each mapping event occurs at a regular time interval of 50 seconds. On each mapping event, we updated the waiting task queue with the new tasks, and monitored the status of pre-scheduled tasks by communicating with LS of each cluster. Upon the completion of a task, we updated the total system utility earned since the last mapping event. Based on the state of the nodes (available or busy) and the wait queue, we ran the heuristics to identify the next set of

tasks for scheduling. We considered dynamic energy as the energy metric for all the heuristics.

6.3. Experimental Results

The duration for each experiment was set to 180 minutes to utilize the generated three hour workload. We repeated the experiment four times for each of the heuristics using the same workload. We then calculated the average utility for each heuristic. We calculated the maximum limit on dynamic energy by running the Max UPR heuristic for 180 minutes. We defined an energy constrained environment by setting a new threshold relative to the maximum limit on dynamic energy. We used 85% and 70% as the two thresholds to compare the behavior of all the heuristics under different energy constraints. We terminated the experiment when either the time limit reached the 180 minute mark or dynamic energy consumption exceeded the energy constraint set for the experiment. Figure 16 shows the plot of utility earned for the EASY Backfilling, Max UPR, Max UPE, and the Event-Based Metaheuristic under three energy constraints. Task selection criterion for the EASY Backfilling is independent from the utility of the task. Therefore, the utility earned by the EASY Backfilling is less than the utility based heuristics for all energy constraint levels. With 100% energy constraint, we see the utility earned by Max UPR, Max UPE, and the Event-Based Metaheuristic is approximately the same. As the energy constraint becomes tighter, fewer tasks get completed resulting in a reduction in the total utility earned by all heuristics. Max UPE and the Event-Based Metaheuristic (switches between Max UPR and Max UPE), under tighter energy constraints, behaved similarly and resulted in around 20% and 18% increase in total utility earned compared to the Max UPR heuristic for the 85% and 70% energy constraints, respectively. With the 100% energy constraint, Max UPR, the Event-Based Metaheuristic, and EASY Backfilling consumed approximately all of the allocated energy budget, but Max UPE consumes only 92% of the energy budget.

We performed simulations for a system with only one P-state option for each cluster to better match these experiments. In these simulations, the relative performance of the heuristics was very similar to these experimental results. For example, in an unconstrained system the simulations showed that Max UPE, Max UPR, and the Event-Based Metaheuristic earned similar utility. When an energy constraint was added, Max UPE and the Event-Based Metaheuristic earned the most utility. The similarity of these results suggests that the assumptions we made in designing our simulations (e.g., deterministic execution times) do not significantly alter our results and that our conclusions about the relative performance of each heuristic from these results apply to actual systems.

7. 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., [8, 9, 28, 29, 30]). 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 [28], the authors designed an iterative Tabu search algorithm to improve the fairness of Conservative Backfilling. The Conservative Backfilling heuristic was also modified in [29] 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 (e.g., [31, 32]). This is done through surveying the literature in [31] and through the development of a framework for measuring supercomputer productivity in [32]. 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 [33, 34, 35, 36]. Differences between these works and ours include that [33, 34, 36] do not consider heterogeneity and [36] does not consider parallel tasks.

The authors of [37] 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 [34], 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 [34] 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 of 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.

In [4], a technique called Incremental Static Voltage Adaption (ISVA) is proposed and evaluated. This technique attempts to minimize makespan under an energy constraint. First, ISVA builds a schedule (using any scheduling technique) without an energy constraint using the minimum voltage possible to execute each task. This schedule is then modified by increasing the voltage used

to execute specific tasks to reduce the makespan. This work differs from ours in several significant ways. First, this work considers a static DAG scheduling problem, while our study focuses on scheduling dynamically arriving tasks with no precedence constraints. In addition, this work does not consider utility, and instead uses makespan as the performance measure.

8. Conclusion and Future Work

We designed and evaluated the performance of several utility-aware resource allocation heuristics (Max Util, Max UPT, Max UPR, Max UPE, and two metaheuristics), and associated dropping and filtering techniques. Performance 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 novel concept of place-holders that we presented, 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. In addition, both of our metaheuristics, the Event-Based Metaheuristic and the Task-Based Metaheuristic, were able to earn utility greater than or equal to all other heuristics in environments where there was a steady rate of task arrivals regardless of the energy constraint. In environments with a highly variable task arrival pattern, the Max UPE heuristic performed best in the energy constrained environment. In addition, it is worth noting that although energy consumption is limited by a constraint in this work and is not a goal for optimization, the Max UPE heuristic often earns utility equal to the other best performing heuristics while consuming significantly less energy.

A topic that we are interested in exploring in the future is employing the concept of preemption in an environment where the system performance measure is based on a time varying utility. 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.

The energy filters presented in this paper could be used to dynamically control the energy consumption of the system by adjusting the energy budget or energy-per-resource budget throughout the day. This could be used to more effectively manage system resources in environments with time-of-use pricing (i.e., environments where energy prices change throughout the day).

Our metaheuristics that switch between Max UPR and Max UPE perform well in environments where tasks arrive at a steady rate, but when the task arrival rate varies significantly throughout the day the metaheuristics do not perform as well as Max UPE when there is a tight energy constraint. Currently, the metaheuristics attempt to guide the system to consume energy at a constant rate throughout the day. It would be interesting to modify the metaheuristics so that the goal for the system's rate of energy consumption is the same as the expected arrival pattern of tasks. This may result in performance of the metaheuristics that matches the best of Max UPR or Max UPE in all environments instead of just environments where tasks arrive at a steady rate.

Acknowledgments

This manuscript has been administered 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). This research was also supported by the National Science Foundation (NSF) under grant number CCF-1302693. This work utilized CSU's ISTeC Cray system, which is supported by NSF under grant number CNS-0923386.

References

[1] B. Khemka, R. Friese, L. D. Briceo, H. J. Siegel, A. A. Maciejewski, G. A. Koenig, C. Groer, G. Okonski, M. M. Hilton, R. Rambharos, S. Poole, Utility functions and resource management in an oversubscribed heterogeneous computing environment, IEEE Transactions on Computers 64 (8) (Aug. 2015) pp. 2394–2407.

[2] P. Bohrer, E. N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, R. Rajamony, The case for power management in web servers, Power Aware Computing, Series in Computer Science (2002) pp. 261–289.

[3] 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, Aug. 2010, pp. 31–45.

[4] I. Ahmad, R. Arora, D. White, V. Metsis, R. Ingram, Energy-constrained scheduling of DAGs on multi-core processors, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 592–603.

[5] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, T. Mezzacappa, P. Moin, M. Norman, R. Rosner, V. Sarkar, A. Siegel, F. Streitz, A. White, M. Wright, The opportunities and challenges of exascale computing, Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee at the US Department of Energy Office of Science (2010).

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

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

[8] D. A. Lifka, The ANL/IBM SP scheduling systems, Job Scheduling Strategies for Parallel Processing, Vol. 949 of Lecture Notes in Computer Science (1995) pp. 295–303.

[9] A. W. Mualem, D. G. Feitelson, Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling, IEEE Transactions on Parallel and Distributed Systems 12 (6) (June 2001) pp. 529–543.

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

[11] D. Machovec, B. Khemka, S. Pasricha, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, M. Wright, M. Hilton, R. Rambharos, N. Imam, Dynamic resource management for parallel tasks in an oversubscribed energy-constrained heterogeneous environment, in: 25th Heterogeneity in Computing Workshop (HCW 2016), 2016 International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2016), May 2016, pp. 67–78.

[12] 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, accessed: 2015-08-30 (Sep. 2004).
URL [Online]:http://www.intel.com/content/dam/www/public/us/en/documents/articles/acpi-config-power-interface-spec.pdf

[13] H. Barada, S. M. Sait, N. Baig, Task matching and scheduling in heterogeneous systems using simulated evolution, in: 10th IEEE Heterogeneous Computing Workshop (HCW 2001), Apr. 2001, pp. 875–882.

[14] A. Ghafoor, J. Yang, A distributed heterogeneous supercomputing management system, IEEE Computer 26 (6) (June 1993) pp. 78–86.

[15] A. Khokhar, V. K. Prasanna, M. E. Shaaban, C. Wang, Heterogeneous computing: Challenges and opportunities, IEEE Computer 26 (6) (June 1993) pp. 18–27.

[16] Colorado State University ISTeC Cray high performance computing system, [Online]: http://istec.colostate.edu/activities/cray, accessed: 2016-11-10.

[17] O. H. Ibarra, C. E. Kim, Heuristic algorithms for scheduling independent tasks on non-identical processors, Journal of the ACM 24 (2) (Apr. 1977) pp. 280–289.

[18] 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, 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 61 (6) (June 2001) pp. 810–837.

[19] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, R. F. Freund, Dynamic mapping of a class of independent tasks onto heterogeneous computing systems, Journal of Parallel and Distributed Computing 59 (2) (Nov. 1999) pp. 107–131.

[20] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, R. F. Freund, Dynamic mapping of a class of independent tasks onto heterogeneous computing systems, Journal of Parallel and Distributed Computing, Special Issue on Software Support for Distributed Computing 59 (2) (Nov. 1999) pp. 107–131.

[21] S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, S. Ali, Representing task and machine heterogeneities for heterogeneous computing systems, Tamkang Journal of Science and Engineering, Special Tamkang University 50th Anniversary Issue 3 (3) (Nov. 2000) pp. 195–207, invited.

[22] A. B. Downey, A model for speedup of parallel programs, Technical Report UCB/CSD-97-933, EECS Department, University of California, Berkeley (1997).

[23] Parallel workloads archive, [Online]: http://www.cs.huji.ac.il/labs/parallel/workload/, accessed: 2015-11-10.

[24] D. G. Feitelson, D. Tsafrir, D. Krakov, Experience with using the parallel workloads archive, Journal of Parallel and Distributed Computing 74 (10) (Oct. 2014) pp. 2967–2982.

[25] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, KVM: The Linux virtual machine monitor, The Linux Symposium 1 pp. 225–230, July 2007.

[26] BladeCenter Advanced Managment Module (User's Guide), [Online]: https://publib.boulder.ibm.com/infocenter/bladectr/documentation/topic/com.ibm.bladecenter.advmgtmod.doc/kp1bb_pdf.pdf, accessed: 2015-08-30.

[27] NAS parallel benchmarks, NAS-NPB, [Online]: https://www.nas.nasa.gov/publications/npb.html, accessed: 2015-08-30.

[28] D. Klusaccek, H. Rudova, Performance and fairness for users in parallel job scheduling, Job Scheduling Strategies for Parallel Processing, Vol. 7698 of Lecture Notes in Computer Science (2012) pp. 235–252.

[29] A. Mishra, S. Mishra, D. S. Kushwaha, An improved backfilling algorithm: SJF-B, International Journal on Recent Trends in Engineering and Technology 5 (1) (Mar. 2011) pp. 78–81.

[30] Y. Yuan, Y. Wu, W. Zheng, K. Li, Guarantee strict fairness and utilize prediction better in parallel job scheduling, IEEE Transactions on Parallel and Distributed Systems 25 (4) (Apr. 2014) pp. 971–981.

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

[32] M. Snir, D. A. Bader, A framework for measuring supercomputer productivity, International Journal of High Performance Computing Applications 18 (4) (Nov. 2004) pp. 417–432.

[33] E. Jensen, C. Locke, H. Tokuda, A time-driven scheduling model for real-time systems, in: IEEE Real-Time Systems Symposium, 1985, pp. 112–122.

[34] P. Li, B. Ravindran, S. Suhaib, S. Feizabadi, A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems, IEEE Transactions on Software Engineering 30 (9) (Sep. 2004) pp. 613–629.

[35] C. B. Lee, 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), 2007, pp. 107–116.

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

[37] G. Sabin, R. Kettimuthu, A. Rajan, P. Sadayappan, Scheduling of parallel jobs in a heterogeneous multi-site environment, Job Scheduling Strategies for Parallel Processing, Vol. 2862 of Lecture Notes in Computer Science (2003) pp. 87–104.

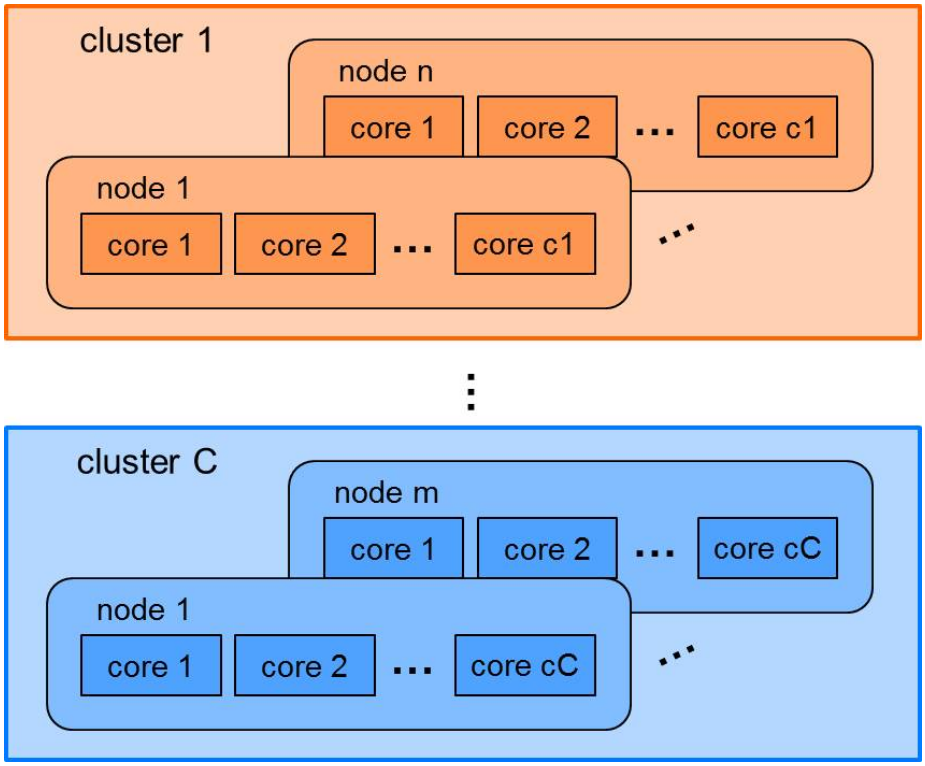


Figure 1: A compute system composed of C clusters. Cluster 1 has n nodes and cluster C has m nodes.

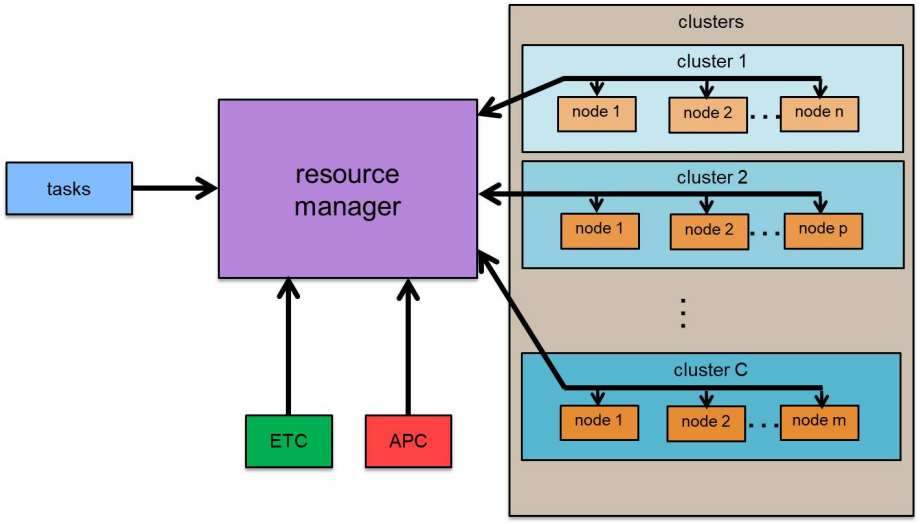


Figure 2: Flow for the proposed resource manager. Tasks enter the resource manager and are mapped to the nodes of clusters. Each task is mapped to the nodes of a single clusters.

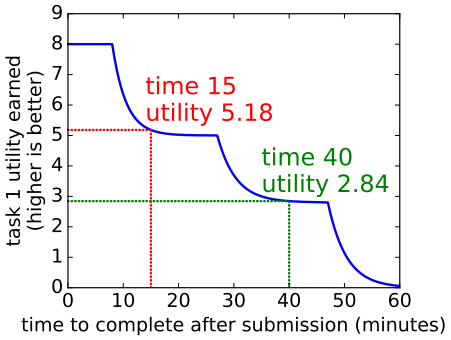
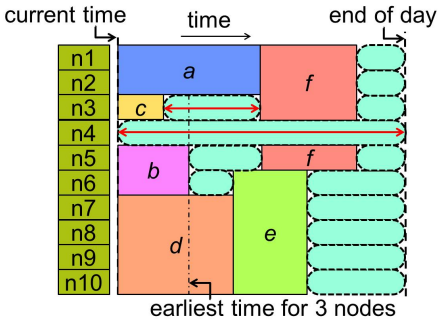
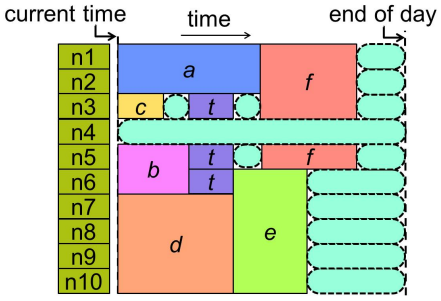


Figure 3: An example of a utility function for task 1. If task 1 completes at time 15, it earns 5.18 utility. If task 1 complete at time 40, it earns 2.84 utility.



(a)



(b)

Figure 4: An example of a mapping on a cluster with ten nodes. The colored rectangles represent different tasks, and the rounded rectangles represent voids where new tasks can be inserted. (a) State of a cluster before assigning task where the first time available to schedule the task is shown. (b) the selected nodes for task t Note that n4 is not chosen due to the second tiebreaking criteria described in Subsection 3.7.

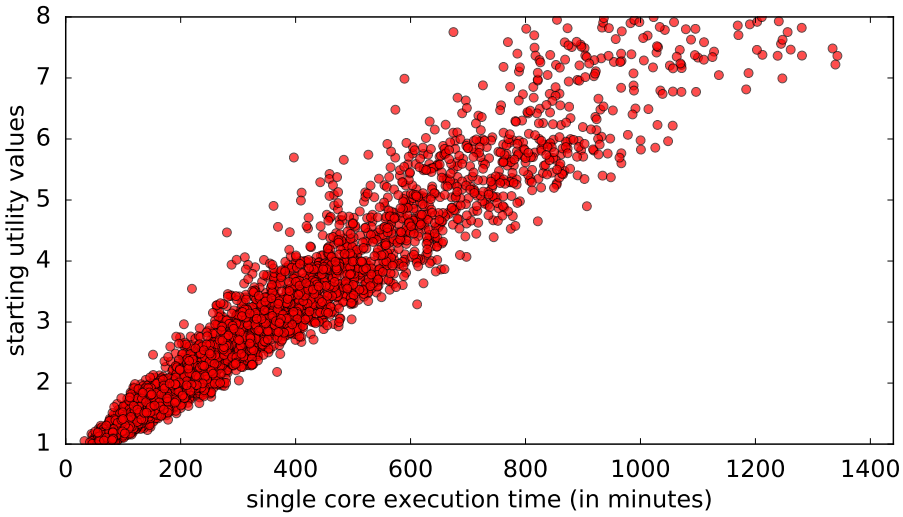


Figure 5: The correlation between single core execution time and starting utility for the 48 simulation scenarios, each with 100 task types, for a total of 4,800 points.

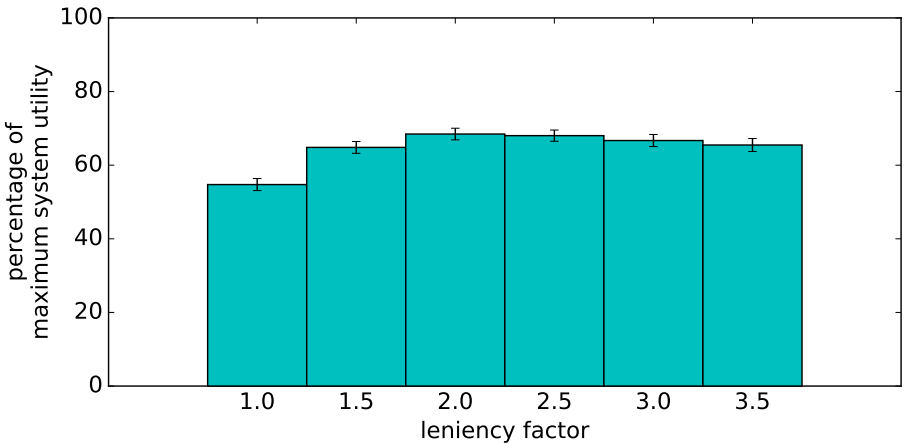


Figure 6: A range of energy leniency factors using energy-per-task filtering for the Max UPR with place-holders heuristic.

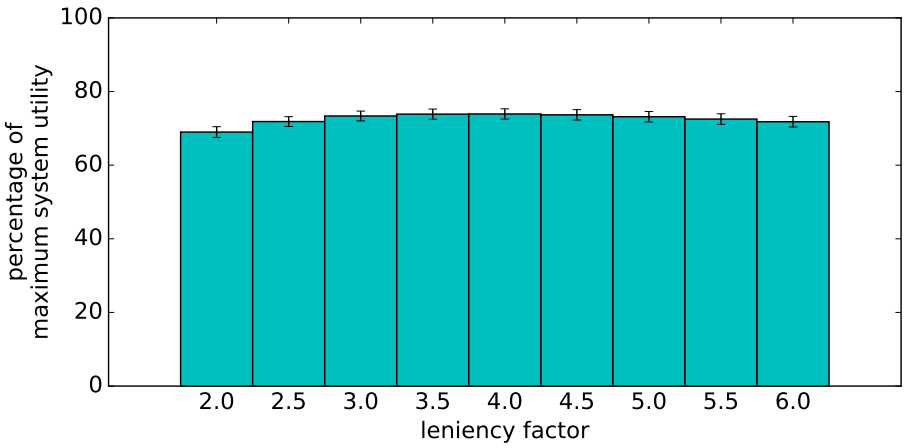
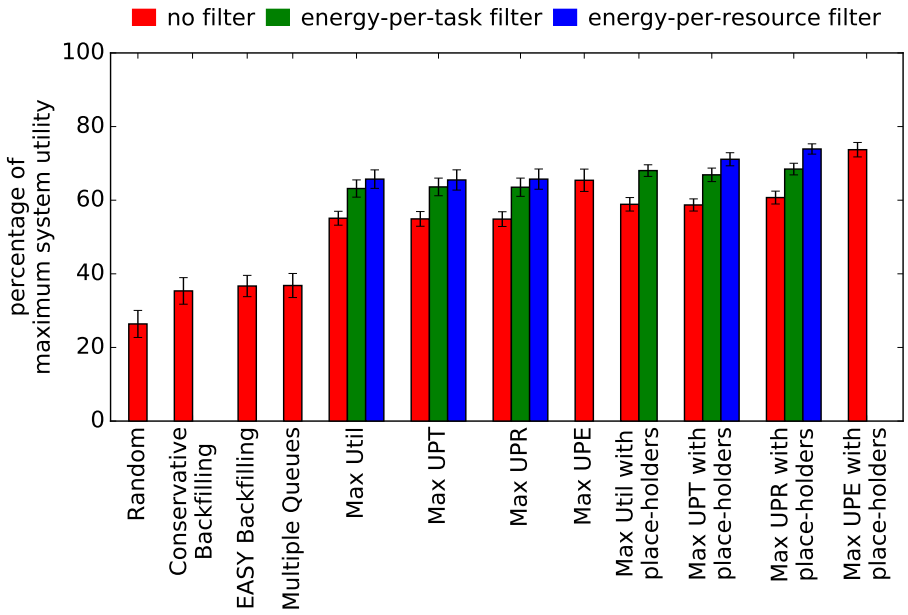
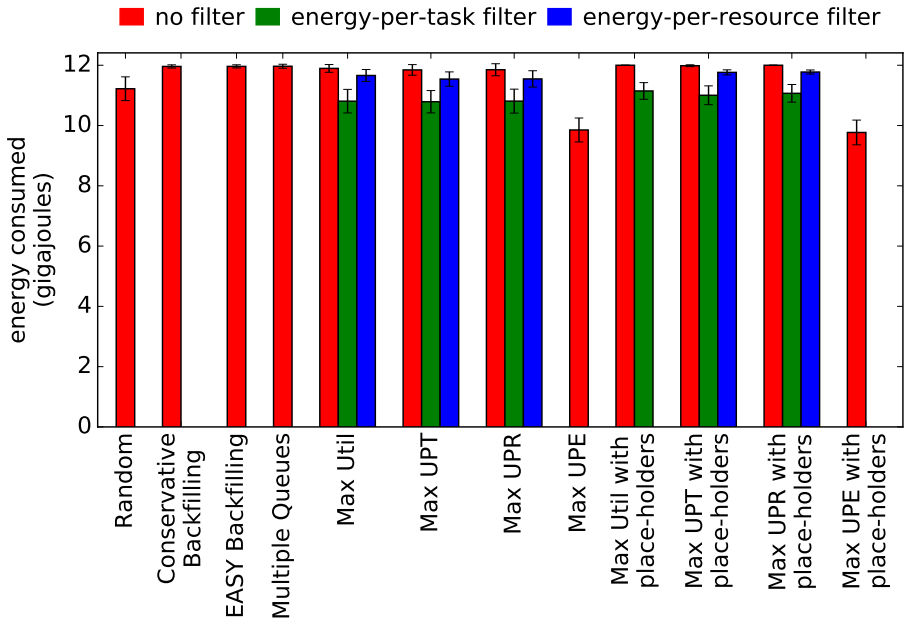


Figure 7: A range of energy leniency factors using energy-per-resource filtering for the Max UPR with place-holders heuristic.

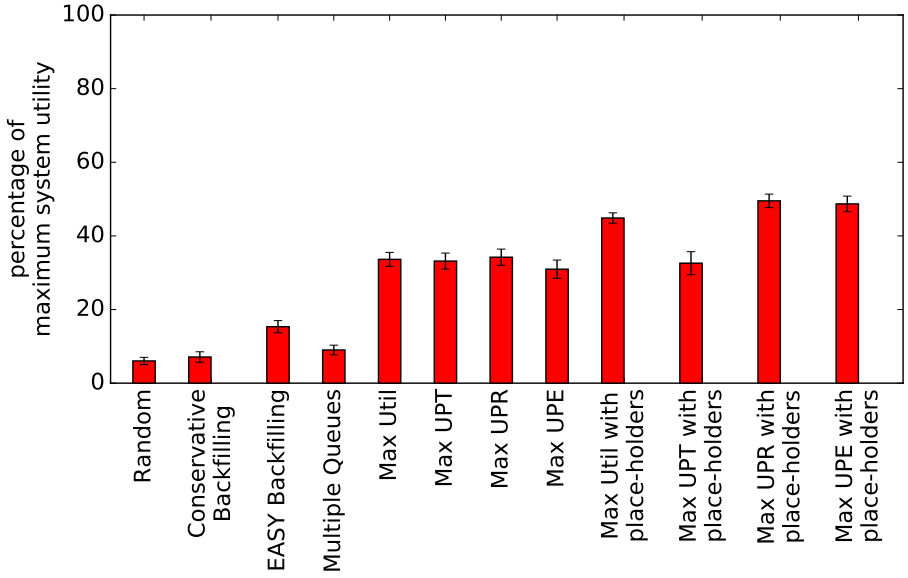


(a)

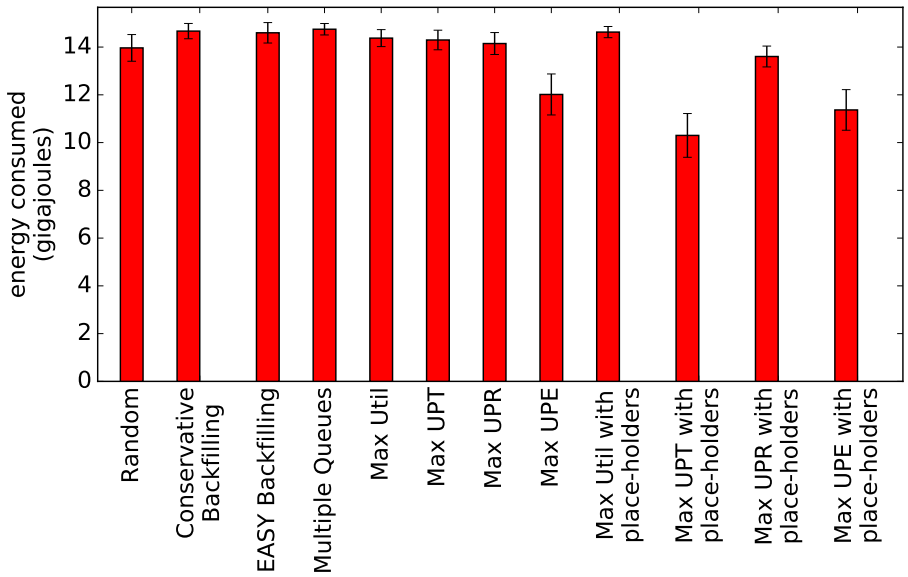


(b)

Figure 8: 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. (a) The percentage of maximum utility earned with 95% confidence intervals. (b) The energy consumption of each heuristic with 95% confidence intervals.

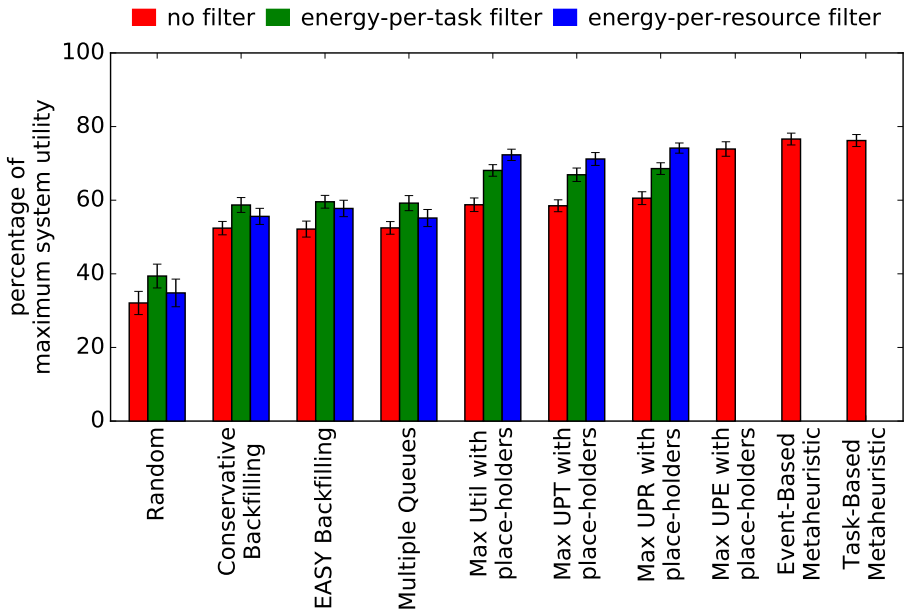


(a)

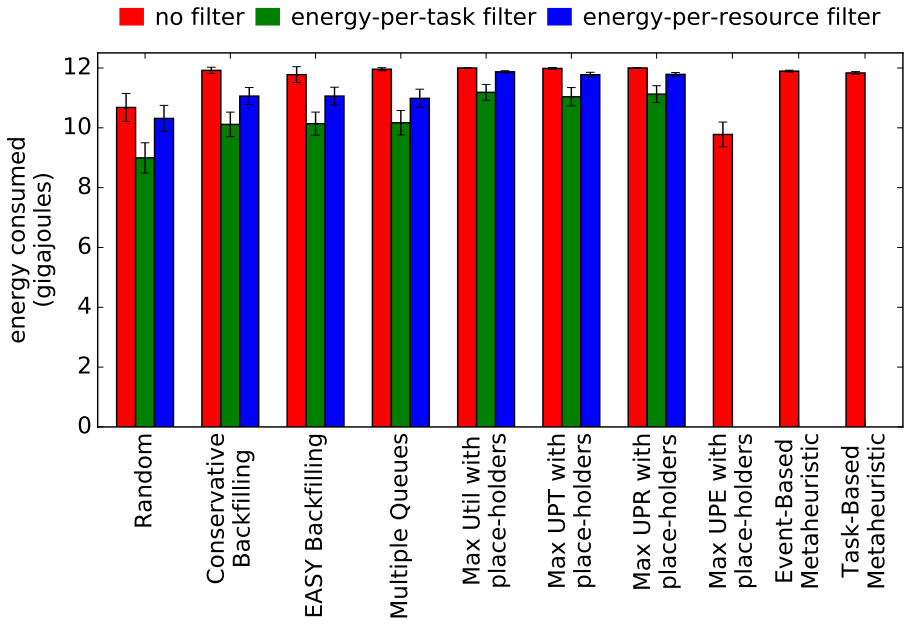


(b)

Figure 9: 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. (a) Percentage of maximum utility earned with 95% confidence intervals. (b) Energy consumption of each heuristic with 95% confidence intervals.



(a)



(b)

Figure 10: Results for a mean of 5,000 tasks arriving per day. A comparison of the two metaheuristics with the heuristics from 3.6, where the utility-based heuristics all use place-holders and all heuristics use a dropping threshold of 0.5. (a) Percentage of maximum utility earned with 95% confidence intervals. (b) Energy consumption of each heuristic with 95% confidence intervals.

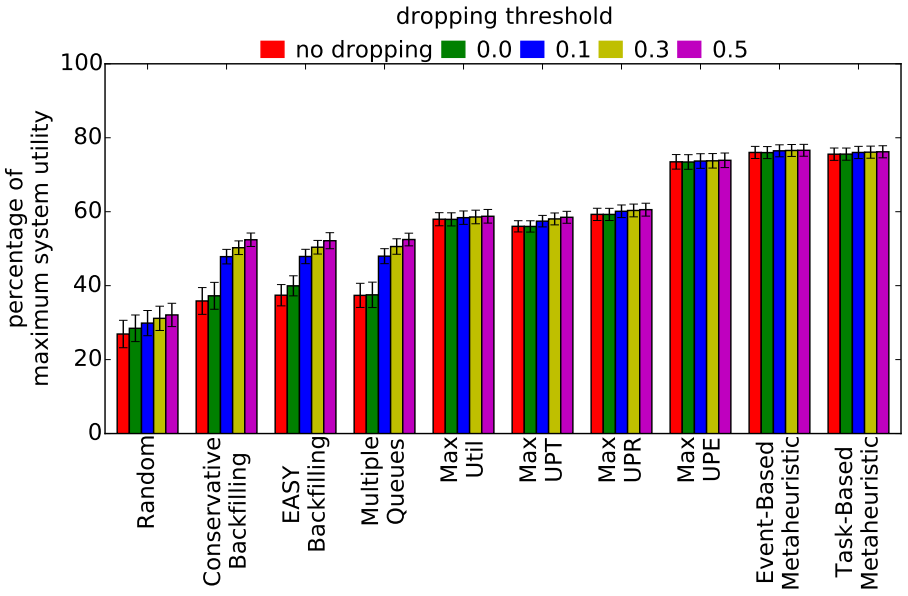
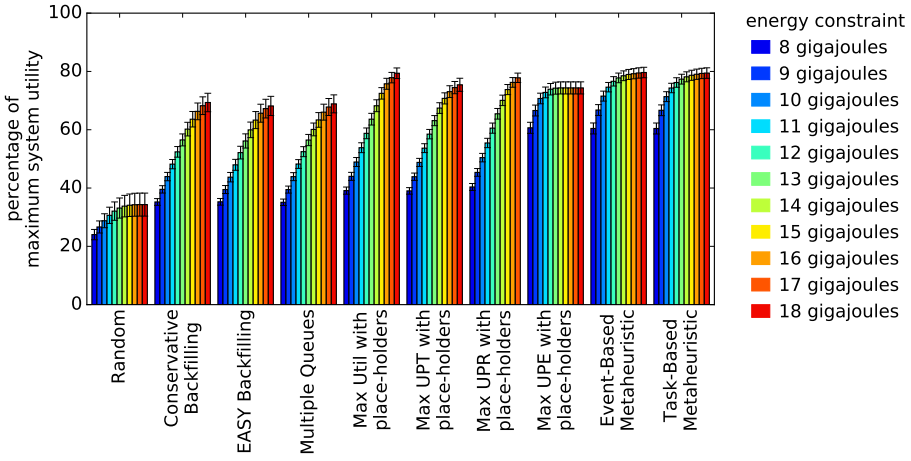
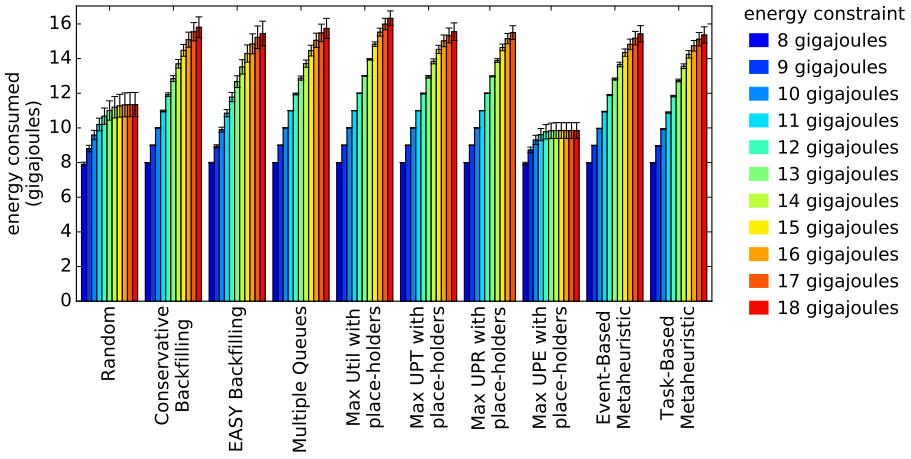


Figure 11: Results for a mean of 5,000 tasks arriving per day, where the dropping threshold is varied, the energy constraint is 12 gigajoules, and none of the heuristics utilize the energy filtering techniques. The percentage of maximum utility earned for each environment with 95% mean confidence intervals is shown.

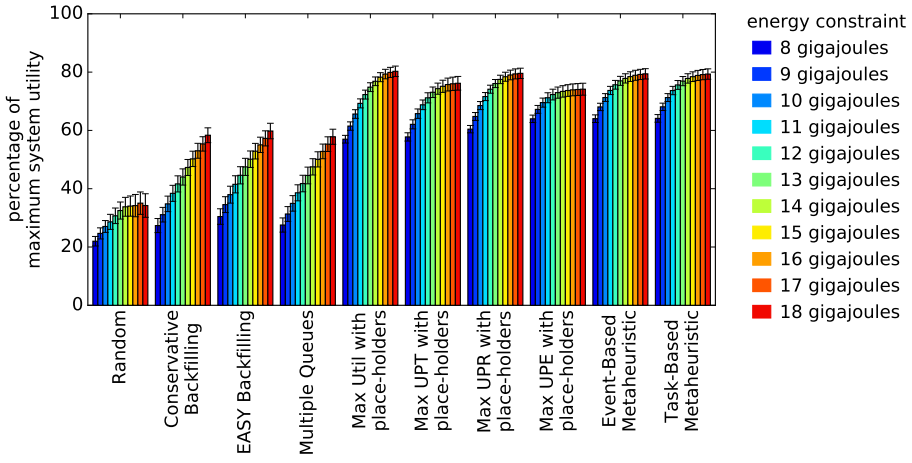


(a)

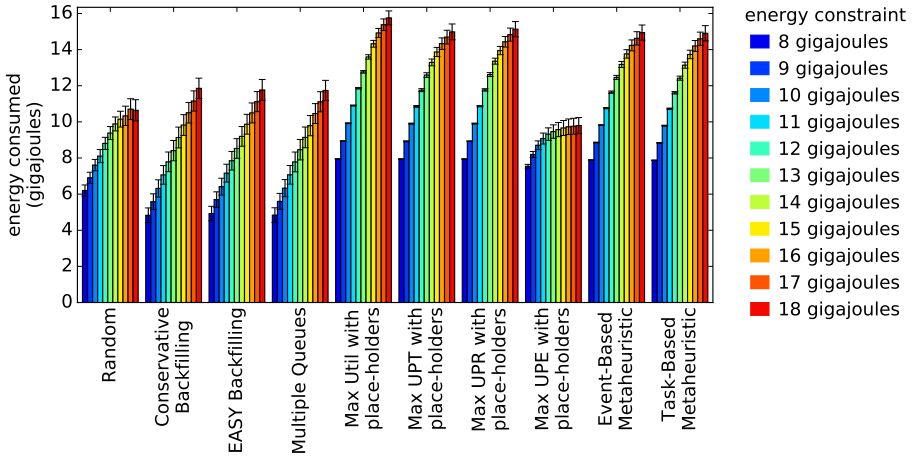


(b)

Figure 12: Results for a mean of 5,000 tasks arriving per day where the energy constraint is varied from 8 gigajoules to 18 gigajoules. None of the heuristics utilize the energy filtering techniques. (a) Percentage of maximum utility for a variety of energy constraints with no energy filter with 95% mean confidence intervals. (b) Energy consumption for a variety of energy constraints with no energy filter with 95% mean confidence intervals.

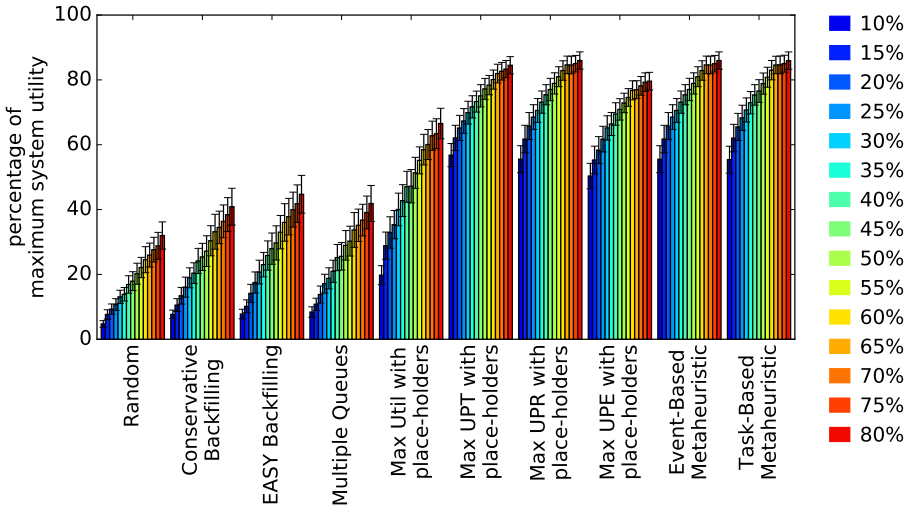


(a)

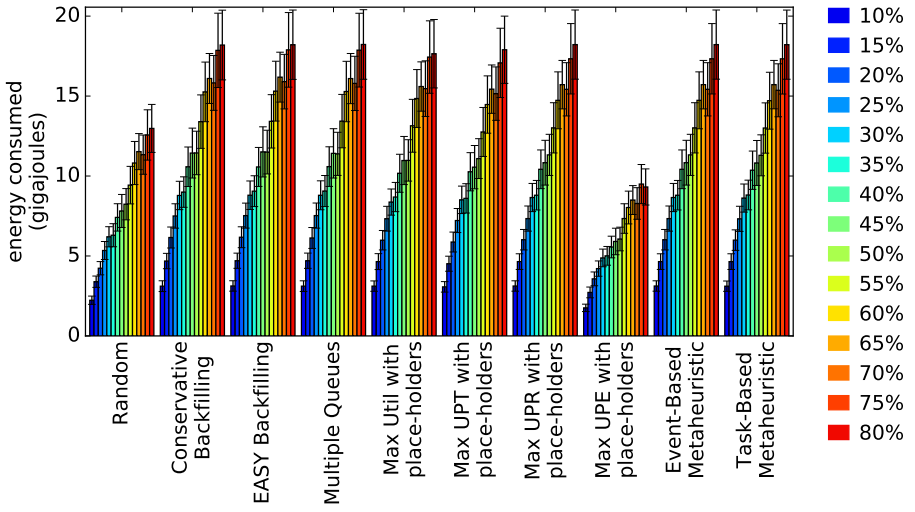


(b)

Figure 13: Results for a mean of 5,000 tasks arriving per day where the energy constraint is varied from 8 gigajoules to 18 gigajoules. All of the heuristics make use of the energy-per-resource filter with a leniency factor of 4.0. (a) Percentage of maximum utility earned for each environment with 95% mean confidence intervals. (b) Energy consumption for each environment with 95% mean confidence intervals.

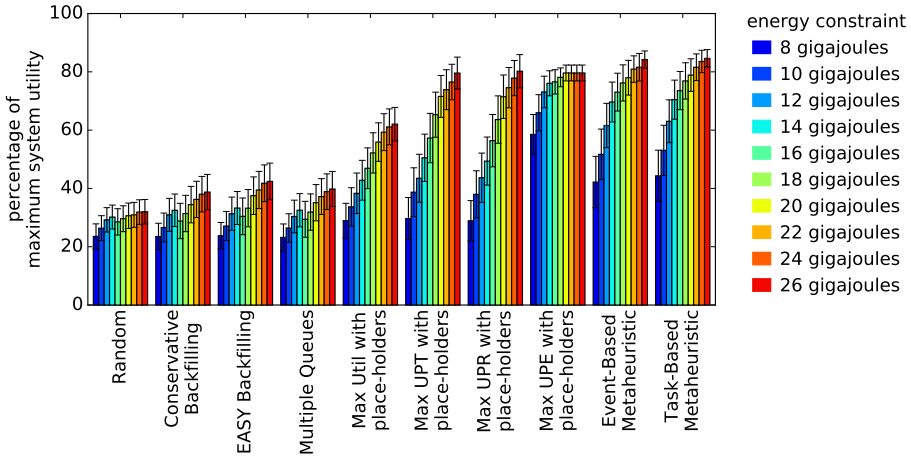


(a)

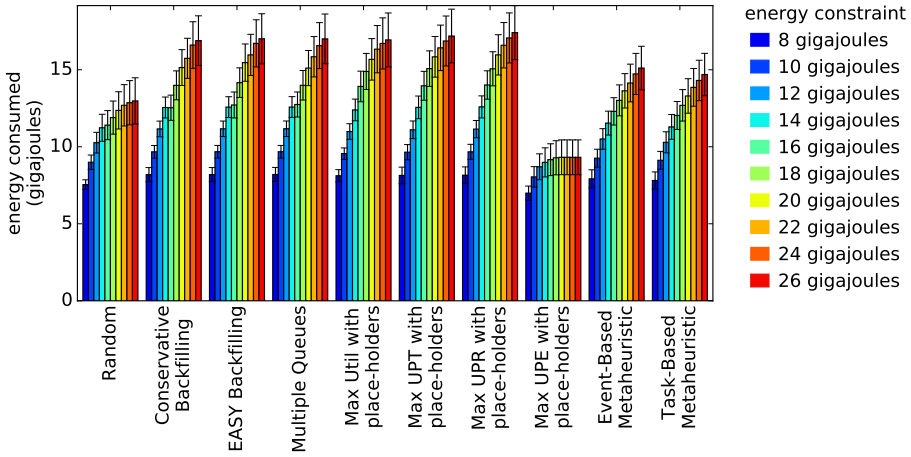


(b)

Figure 14: Results when tasks are based on a trace from the Curie supercomputer and there is no energy constraint. In these environments, the size of the system is varied from 10% to 80% of the original Curie system. (a) Percentage of maximum utility earned for each environment with 95% mean confidence intervals. (b) Energy consumption for each environment with 95% mean confidence intervals.



(a) Percentage of maximum utility for a variety of energy constraints with a real task trace.



(b) Energy consumption for a variety of energy constraints with a real task trace.

Figure 15: Results when tasks are generated using a trace from the Curie supercomputer and there is a varied energy constraint. In these environments the size of the system is equal to 80% of the original Curie system. (a) Percentage of maximum utility earned for each environment with 95% mean confidence intervals. (b) Energy consumption for each environment with 95% mean confidence intervals.

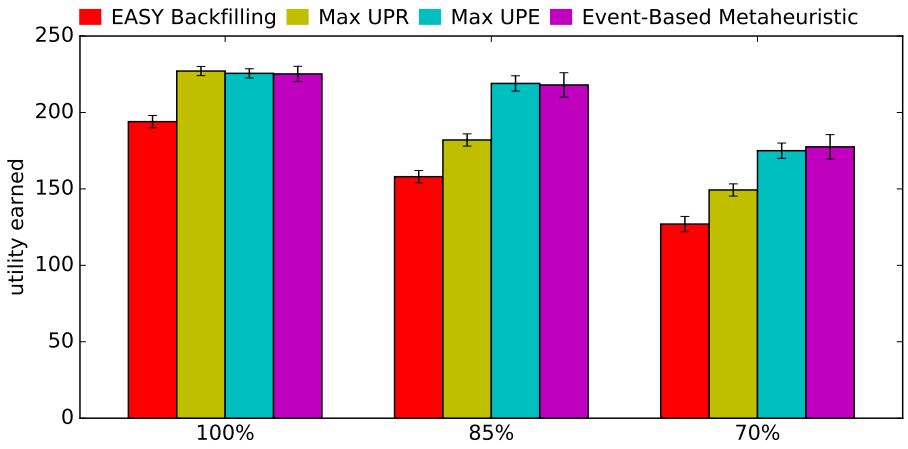


Figure 16: Utility earned versus energy constraint, as percent of maximum allowed, where maximum is 2.8 megajoules. The standard deviation is shown for each bar.

Table 1: An example of an ETC matrix that specifies execution time for task type and number of nodes for a given cluster and P-state.

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		

Table 2: Priority and urgency table.

priority level	starting 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%

Table 3: 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

Table 4: Cluster configuration showing heterogeneity.

cluster	cores per node	RAM size (GB) per node	total number of nodes	maximum allowable operating frequency (MHz)	maximum C-state
cluster 1	1	1	16	1596	C1
cluster 2	2	1	8	1862	C6
cluster 3	4	1.5	4	2261	C1
cluster 4	8	2	2	2660	C6

Table 5: Estimated time to compute (ETC) matrix.

task	core requirement	execution time on cluster 1 (sec)	execution time on cluster 2 (sec)	execution time on cluster 3 (sec)	execution time on cluster 4 (sec)
IS.8	8	1,675	750	552	430
MG.8	8	1,100	712	n.a.	n.a.
MG.16	16	702	500	380	326
IS_CG.8	8	1,039	900	700	1,366

Table 6: Estimated energy to compute (EEC) matrix.

task	energy on cluster 1 (J)	energy on cluster 2 (J)	energy on cluster 3 (J)	energy on cluster 4 (J)
IS.8	25,962	35,250	17,388	28,380
MG.8	27,500	41,296	n.a.	n.a.
MG.16	34,398	55,000	30,400	50,530
IS_CG.8	23,897	47,700	27,650	105,865