DISSERTATION

DYNAMIC RESOURCE MANAGEMENT IN HETEROGENEOUS SYSTEMS: MAXIMIZING UTILITY, VALUE, AND ENERGY-EFFICIENCY

Submitted by

Dylan Machovec

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2021

Doctoral Committee:

Advisor: H. J. Siegel

Anthony A. Maciejewski
Sudeep Pasricha
Patrick Burns

ABSTRACT

DYNAMIC RESOURCE MANAGEMENT IN HETEROGENEOUS SYSTEMS:
MAXIMIZING UTILITY, VALUE, AND ENERGY-EFFICIENCY

The need for high performance computing (HPC) resources is rapidly expanding throughout many technical fields, but there are finite resources available to meet this demand. To address this, it is important to effectively manage these resources to ensure that as much useful work as possible is completed. In this research, HPC systems executing parallel jobs are considered with and without energy constraints. Additionally, the case where preemption is available is considered for HPC systems executing only serial jobs. Dynamic resource management techniques are designed, evaluated, and compared in heterogeneous environments to assign jobs to HPC nodes. These techniques are evaluated based on system-wide performance measures (value or utility), which quantify the amount of useful work accomplished by the HPC system. Near real-time heuristics are designed to optimize performance in specific environments and the best performing techniques are combined using intelligent metaheuristics that dynamically switch between heuristics based on the characteristics of the current environment. Resource management techniques also are designed for the assignment of unmanned aerial vehicles (UAVs) to surveil targets, where performance is characterized by a value-based performance measure and each UAV is constrained in its total energy consumption.

ACKNOWLEDGMENTS

This dissertation was possible thanks to the guidance and support of many people. I would like to specially thank my advisor, Prof. Howard J. Siegel, for the immense time, financial support, and patience he has given me. His knowledge, experience, and valuable lessons have shaped me as a researcher over these last six years. Thank you for encouraging me to continue pressing forward and for seeing this through to the end.

I thank Prof. Sudeep Pasricha and Prof. Anthony A. Maciejewski for serving on my Ph.D. committee. The tremendous support and patience they have given me over these six years have made it possible for me to complete this Ph.D. and have greatly improved the quality of my research. Additionally, thanks to Prof. Patrick J. Burns for also serving on my Ph.D. committee and providing additional insights and perspective to my research.

I thank the remaining co-authors of my publications for their valuable comments and contributions on this research: Ali Akoglu, Christopher Blandin, Jim Crowder, Farah Fargo, Ryan Friese, Salim Hariri, Marcia Hilton, Neena Imam, Bhavesh Khemka, Gregory A. Koenig, Nirmal Kumbhare, Ahmed Louri, Thomas Naughton, Rajendra Rambharos, Cihan Tunc, and Michael Wright. Additionally, thanks to Daniel Dauwe, Ninad Hogade, and John Carbone.

I also thank my parents, Chuck Machovec and Risa Machovec, who have invested so much time and energy while I pursued my Ph.D. I would not have achieved what I have without their loving support.

The work in this dissertation was partly supported by National Science Foundation (NSF) research projects NSF CNS-1624668, SES-1314631, CCF-1302693, and DUE-1303362. Furthermore, this work utilized Colorado State University's ISTeC Cray system, which is

supported by the NSF under grant number CNS-0923386. 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).

DEDICATION

To my parents for their endless support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction and Overview

The use of high performance computing (HPC) systems is rapidly increasing in many technical fields. This includes the need for results from large-scale meteorological models for predicting climate changes or local weather patterns and bioinformatics pipelines that can be used to understand DNA sequences that are too complex to process without the aid of HPC resources. Because HPC resources produce such important results, it is critical to ensure that they are used as efficiently as possible. This is accomplished using resource management techniques for HPC systems, which must assign jobs to HPC nodes and schedule their execution in ways that maximize the amount of useful work completed. To do this, it is necessary to define metrics that can represent the amount of useful work completed by a system. In this research, value-based and utility-based metrics are used to quantify the worth of completing individual jobs. This is used for the comparison, analysis, and evaluation of intelligent resource management techniques.

Balancing the performance needs of users with the energy efficiency desired by HPC system owners is problematic because improving user performance often costs more energy. To address this, a Value of Service (VoS) metric is defined in Chapter 2, which combines value functions of completion time after arrival and energy consumption to obtain a single system-wide performance measure. The value of energy consumption is modified based on the time of day to capture the increased need for energy efficiency when usage is at its peak in the middle of the day. Value-based heuristics are designed and compared against common heuristics from the literature using both simulations and experiments.

In the dynamic energy-constrained environments is Chapter 3, utility earned by jobs must be maximized while also saving energy for possible future jobs that have not yet arrived. This requires making intelligent trade-offs between utility earned and energy consumed. A variety of heuristics are designed, analyzed, and compared in energy-constrained environments and it is shown that certain heuristics perform well in tightly energy-constrained environments, while others perform best when energy is not a significant concern. Based on this, two metaheuristics that intelligently switch between energy-efficient and utility-focused behavior are designed. Additionally, even with energy-efficient heuristics, it is possible that energy would be consumed too quickly in tightly constrained environments. To address this, energy filtering techniques are also considered, which limit the options available to the heuristics to prevent energy from being consumed too quickly before the day ends. Simulation results demonstrate that the energy filtering techniques help significantly in tightly constrained environments and that the metaheuristics perform well regardless of the energy-constraint, which means that system owners do not need to determine if their system requires energy-efficient heuristics or heuristics that prioritize utility because the metaheuristic handles both cases effectively.

Sometimes, HPC systems will have urgent jobs of high importance. When a system is oversubscribed, it may be necessary to preempt running jobs to meet the needs of the urgent jobs. Chapter 4 explores adjusting utility functions to capture the urgency and importance of these jobs. New preemption-capable heuristics are designed, which are evaluated and compared with simulations against utility-based heuristics that are known to perform well in environments where preemption is not possible. It is shown that the best preemption-capable heuristics always perform at least as well as the heuristics that do not consider preemption, but in some cases where the deadlines of urgent jobs would be missed, having preemption available results in significantly

more system utility earned. An important finding for deciding whether to deploy a preemption-capable scheduler is that the maximum amount of time high priority jobs will need to wait before nodes become idle should be considered. If an environment where users are submitting large batches of similar jobs all at once, it will be possible for all nodes to be busy until a batch finishes, resulting in a need for preemption. If job start times are distributed more evenly, then new nodes will frequently become available allowing high priority jobs to avoid long queue times without preemption.

Resource management is also critical in fields outside of HPC. For example, in active battlefield scenarios, the creation of mission schedules that assign unmanned aerial vehicles (UAVs) to surveil specific targets is an area of increasing interest. Like the HPC environments, it is important to determine a useful metric for these problems to allow comparison of possible solutions.

A new value-based performance measure is applied to a UAV surveillance mission scheduling problem in Chapter 5. To study this problem, it is necessary to develop a realistic model for UAV surveillance of targets in addition to a framework for generating realistic scenarios to examine with simulations. This enables accurate and precise comparison and evaluation of mission scheduling techniques. In the design of these techniques, many of the lessons learned from HPC resource management can be applied. This includes the creation of a modified version of the metaheuristics from Chapter 3, which balances the energy consumption of individual UAVs with the system-wide surveillance value performance measure. This metaheuristic is further modified using a preemption technique, which allows UAVs to surveil important targets more readily, and a filtering technique. This filtering technique has similarities in concept to the energy filtering in Chapter 3; however, instead of relying on an empirically determined and static leniency factor, it calculates a

dynamic threshold of value divided by energy for each UAV. This dynamic threshold is designed to control the rate at which a UAV consumes its energy while also ensuring that the UAV spends all of its energy before the end of the day. The simulation results show excellent performance of the improved metaheuristic relative to comparison techniques.

Overall conclusions of the research presented in this work are detailed in Chapter 6. In Chapter 7, other directions for potential future research based on this research are discussed.

Chapter 2

Value of Service Based Resource Management for Large-Scale Computing Systems[1]

2.1. Introduction

Management of large-scale computing systems (e.g., clusters, public and private cloud systems, industry and government large scale data centers) requires balancing the performance demand of the end-users and the operational cost for the service provider. It is expected that the energy consumption of a large scale data center with 50,000 computing nodes may go beyond 100 million kwh/year resulting in more than the yearly electricity consumption of an urban population of 100,000's [1-3]. The International Energy Agency states that the total data center electricity consumption is projected to increase to approximately 140 billion kwh/year by 2020, the equivalent annual output of 50 power plants, costing American businesses $13 billion per year in electricity and causing the emission of nearly 150 million metric tons of carbon pollution annually [4, 5]. If cloud resources are managed efficiently, it is expected that the data center electricity consumption could be reduced by 40% [6]. Furthermore, for exascale computing, the power limit is expected to have an upper bound of 20 MW, which requires at least an order-of-magnitude

[1]The material in this chapter appeared in [57]. This work was done jointly with former Ph.D. student Bhavesh Khemka. The full list of co-authors for this work is at [57] and preliminary versions of this work appeared in [20, 21]. This work was partly supported by National Science Foundation (NSF) research projects NSF CNS-1624668, SES-1314631, CCF-1302693, and DUE-1303362. Furthermore, this work utilized Colorado State University's ISTeC Cray system, which is supported by the NSF under grant number CNS-0923386.

improvement in energy efficiency and resource management techniques [7]. Consequently, for large scale systems, energy efficiency without performance loss is a major goal. Considering the complexity and diversity of such systems and their applications, there is a strong interest in the development of new resource management strategies that are performance driven, energy efficient, scalable, and integrate all system components [7].

In this paper, we present a time dependent Value of Service (VoS) metric for resource management algorithms that consider the task submission time (e.g., task arrival during peak versus non-peak periods) in evaluating task's performance value and task's energy consumption value. We define the system VoS for a given workload to be the sum of the values for all tasks executed during a given period of time. The resource utilization of large scale systems typically vary during the day. The electricity consumption of a Google Internet Data Centers (IDC) varies during the day showing peak period or non-peak (idle-periods) [8]. During non-peak periods (such as midnight) the resources are expected to be idle or lightly loaded, whereas during peak times (such as during work hours), resources are expected to be highly utilized. Therefore, the schedulers are typically designed to operate under a worst-case scenario where the system is assumed to be oversubscribed. From the cloud service provider perspective, to reduce the operational cost, one approach is to execute fewer tasks during peak hours than the number of tasks that run during non-peak hours. As an example, Amazon recently introduced a new pricing scheme where the virtual machines (VMs) are discounted up to 75% during the non-peak time periods to increase the system utilization [9]. For some applications, the operational cost is not the main concern because obtaining fast timely results is more critical than the cost of the service, while for other applications postponing the execution of their tasks until the non-peak period is acceptable. For example, a financial application that attempts to predict in real-time stock market trends requires low latency

and high throughput performance when the market is open for trading. Consequently, the scheduler should take these requirements into consideration when allocating resources to applications that maximize the VoS of the overall system operations.

In this paper, we present a novel scheduling methodology that is based on the VoS metric that will be used to guide the assignment of resources to the workload tasks during a given period of time. In our environment, the goal of the scheduler is to allocate the appropriate VM configuration for each task with respect to the number of cores and amount of memory (i.e., one of the allowable resource configurations) that maximizes the VoS for the overall system. To predict the execution time and energy consumption of each task type running on a VM, we use statistical and data mining techniques to model the execution time and energy consumption. We consider an environment where the system workload changes over time to model real-life operational scenarios; during nighttime hours, the system experiences low-utilization, and during daytime hours the system experiences high-utilization (the system can be oversubscribed during peak time periods). Energy costs vary throughout these different periods. Due to the continuous changes in the system utilization and energy costs, it is not guaranteed that all tasks can begin execution immediately upon arrival or all tasks can meet their completion time and energy requirements.

The major contributions of the paper are as follows:

- A time-of-use dependent value based metric to enable scheduling algorithms that take into the value of task completion and energy consumption for a given arrival time period. We combine these task values to form the system VoS performance metric.
- We design a set of resource management heuristics based attempt to maximize the time-of-use system VoS performance metric.
- We evaluate and compare these heuristics using a simulation environment.

- We verify the relative performance of some of these heuristics on a real-testbed using IBM HS22 blade servers.
- A data-driven energy model based on VM resource usage is created to accurately predict the task energy consumption with low computational overhead.

The remaining sections of the paper are organized as follows. In Section 2.2, we discuss the system model and, in Section 2.3, we explain the VoS based scheduling methodology. We describe the VoS metric and scheduling algorithms (resource management heuristics) in Section 2.4. We demonstrate our simulation based evaluation of the performance of the heuristics in Section 2.5. In Section 2.6, we present our experimental environment and results. We review related work in Section 2.7. Finally, Section 2.8 summarizes the paper results.

2.2. System Model

In our model, we target large-scale computing systems composed of homogenous clusters. Tasks are parallel applications and they arrive dynamically with user-specified soft and hard completion thresholds. We couple the completion thresholds with soft and hard energy consumption thresholds imposed by the service provider. The service provider then uses a value function for determining the total value earned depending on these predefined performance and energy consumption thresholds.

Each task can be executed on different VM resource configurations, where a configuration is defined by the number of cores and the amount of memory assigned to the VM. We assume there are fixed number of task types and for each task type, there is a set of *allowable resource configurations*. For each of these allowable resource configurations, we assume that there is a known estimated task execution time and estimated task energy usage that have been benchmarked

a priori. It is common in the resource management literature to assume the availability of such information, e.g., [10-13].

2.3. Value of Service (VoS)

Utility functions ([14-19]) have been shown to be effective metrics in resource management, especially in oversubscribed environments. A primary difference of our VoS metric from utility techniques is the fact that the value metric allows us to consider the value of performing resource management at a particular time of the day or night as well as the actual operational costs of using the allocated resources at a given time. In addition, the VoS metric considers performance and energy efficiency at the same time.

We define the value of a task as a monotonically-decreasing function of a resource management objective (e.g., completion time or energy consumption reduction) that specifies the value earned by completing a task during a given period of time. For example, the energy value earned by reducing energy consumption during a peak energy consumption period is significantly larger than the energy reduction value earned during the idle period. The shape of a particular value function for a given task depends on a set of parameters determined by the end-user in conjunction with the service provider. We illustrate these parameters in Figure 1. The soft threshold parameter specifies the limit on an objective (Th_{soft}) until which the value earned by a task j ($Task_j$) is maximum (v_{max}). Beyond the soft threshold, the value starts decreasing until the objective reaches the hard threshold. The hard threshold (Th_{hard}) specifies a limit for the (v_{min}) to be valid, beyond which zero value is earned. The linear function shown in Figure 1 that models the value change between v_{max} and v_{min} is considered in this study. However, the linear function model can be replaced by other functions based on the user requirements and the operational costs associated with a given data center. In our analysis, we modeled two value functions for two

resource management objectives: (a) completion time (i.e., performance) value function (Figure 2), and (b) energy value function (Figure 3).

The performance value function for a task depends on the completion time during a given period of time. The user defines the soft and hard thresholds for the task completion time. The soft threshold (p_{soft}), as shown in Figure 2, specifies the limit on task completion time until which the performance value earned by the task is maximum. Beyond the soft threshold, the performance

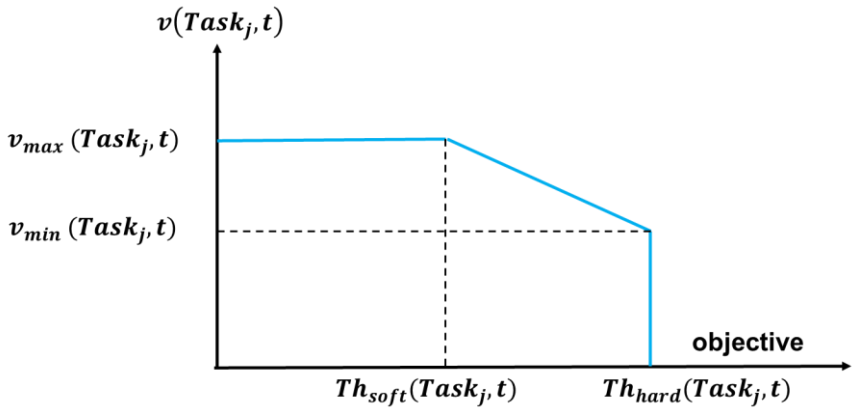


Figure 1. General formulation for value versus objective and thresholds [14].

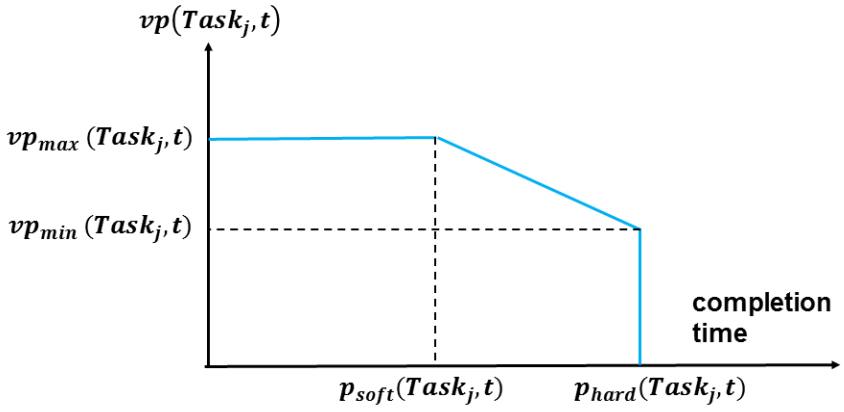
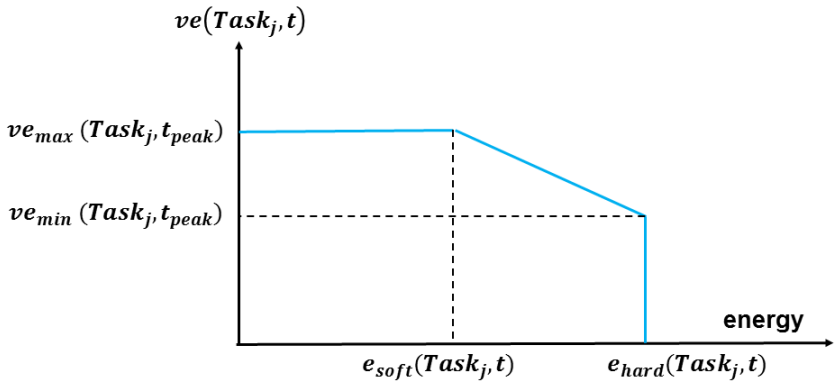


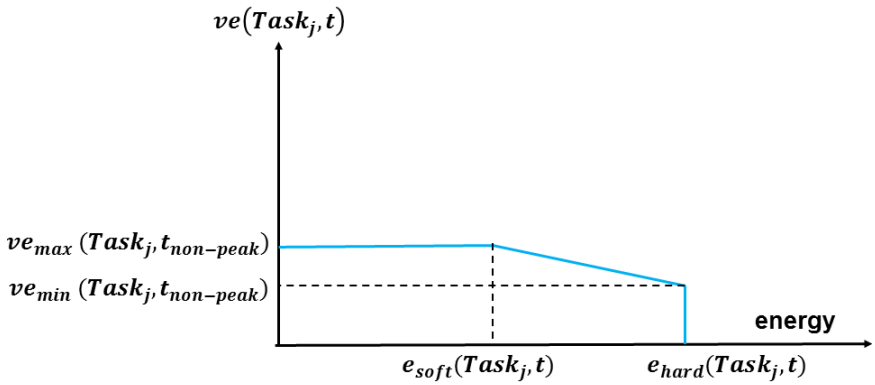
Figure 2. Performance value versus completion time [14].

value starts decreasing until the task completion time reaches the hard threshold (p_{hard}). The hard threshold specifies a limit on the completion time, beyond which zero performance value is earned.

Similarly, the energy value function of a task depends on the energy consumption for its execution during a given period of time as shown in Figure 3 (both Figure 3.a and 3.b). The soft threshold (e_{soft}) specifies the limit on energy consumed by a task until which the energy value earned by task is maximum. Beyond soft threshold, the energy value starts decreasing until the energy consumed by the task reaches the hard threshold. The hard threshold (e_{hard}) specifies the



(a)



(b)

Figure 3. Energy value versus energy consumed. (a) Peak time. (b) Non-peak time.

limit on the energy consumed by the task, beyond which zero energy value is earned. Both thresholds can be defined by the service provider. During the peak time period (i.e., high utilization period), the cloud provider goal is to reduce the operational cost by reducing energy consumption. The difference in energy values for peak (t_{peak}) and non-peak ($t_{non-peak}$) periods are shown in Figure 3.a and Figure 3.b, respectively. As mentioned before, other functions can be used depending on the user requirements and the operational cost of data center resources as a function of time.

If either the performance value or the energy value becomes equal to 0, then the value of the task will also be zero.

We model the VoS as the sum of weighted performance and energy values refined by a relative importance factor among tasks. In this paper, the value function is defined as follows:

- The value ($v(Task_j, t)$) represents the value for completing task "j", which is submitted at time t, with respect to a given objective (e.g., completion time or energy consumption).
- The maximum value ($v_{max}(Task_j, t)$) represents the maximum value that can be achieved by completing task "j." This maximum value will change depending on the submission time t of that task.
- The soft threshold ($Th_{Soft}(Task_j, t)$) defines the limit on the objective value for task "j" to gain the maximum possible value ($v_{max}(Task_j, t)$).
- The hard threshold ($Th_{Hard}(Task_j, t)$) defines the maximum limit on the objective value for task "j" to gain the minimum value ($v_{min}(Task_j, t)$). *After that limit, the value earned by task will be zero.*

- The min value ($v_{min}(Task_j, t)$ represents the minimum positive value that can be achieved by completing the task "j" that is submitted at time t when the objective is equal to the hard threshold (Th_{Hard}).
- The objective ($Obj(Task_j, t)$) represents the completion time or energy consumption of a task "j" that is submitted at time t.

If task "j" does not complete its execution by the $Th_{Hard}(Task_j, t)$ then the value given for task "j" drops to zero. In between $Th_{Soft}(Task_j, t)$ and $Th_{Hard}(Task_j, t)$, the value for task "j" decreases linearly from $v_{max}(Task_j, t)$ to $v_{min}(Task_j, t)$.

In Figure 1, a fixed value is gained till the soft threshold ($Th_{Soft}(Task_j, t)$), which is defined by $v_{max}(Task_j, t)$ (Equation (1)(a)). The task value after the hard threshold ($Th_{Hard}(t_j)$) is zero (Equation (1)(c)). In between these two thresholds, the rate of reduction in the value is defined by Equation (1)(b). Here, we note that for soft and hard threshold constraints and for the performance and energy value functions described below, the measurement units are task completion time (seconds) and energy consumption (Joules), respectively. Table 1 shows the task threshold and performance and energy values.

Table 1. Value function parameters for performance and energy.

Value function parameters	Performance value function parameters	Energy value function parameters
$Obj(Task_j, t)$	$p(Task_j, t)$	$e(Task_j, t)$
$Th_{Soft}(Task_j, t)$	$p_{soft}(Task_j, t)$	$e_{soft}(Task_j, t)$
$Th_{Hard}(Task_j, t)$	$p_{hard}(Task_j, t)$	$e_{hard}(Task_j, t)$
$v(Task_j), t$	$v_p(Task_j, t)$	$v_e(Task_j, t)$
$v_{max}(Task_j, t)$	$vp_{max}(Task_j, t)$	$ve_{max}(Task_j, t)$
$v_{min}(Task_j, t)$	$vp_{min}(Task_j, t)$	$ve_{min}(Task_j, t)$

$$\boldsymbol{a}) \; if \; \Big(0 \leq \; Obj\big(Task_j, t\big) \leq Th_{Soft}\big(Task_j, t\big)\Big) \; then$$

$$v\big(Task_j, t\big) = v_{max}\big(Task_j, t\big),$$

$$\boldsymbol{b}) \; if \; \Big(Th_{Hard}\big(Task_j, t\big) > \; Obj\big(Task_j, t\big) > Th_{Soft}\big(Task_j, t\big)\Big) \; then$$

$$v\big(Task_j, t\big) = \Big(Obj\big(Task_j, t\big) \; - Th_{Hard}\big(Task_j, t\big)\Big)$$

$$* \; \frac{v_{max}\big(Task_j, t\big) \; - \; v_{min}\big(Task_j, t\big)}{Th_{Soft}\big(Task_j, t\big) - Th_{Hard}\big(Task_j, t\big)} + v_{min}\big(t_j\big),$$

$$\boldsymbol{c}) \; else \; if \; \Big(Obj\big(t_j\big) \geq Th_{Hard}\big(Task_j, t\big)\Big) \; then,$$

$$v\big(Task_j, t\big) = 0. \tag{1}$$

Task value ($V(Task_j, t)$) represents the total value earned by completing task *j* submitted at time t. It is the weighted sum of earned performance and energy values, as shown in Equation 2. The w_p and w_e coefficients are used for adjusting the weight given to the performance and energy values. The importance factor $\gamma^{(Task_j)}$ expresses the relative importance among tasks. Also, as mentioned earlier, if either the performance function or energy function is 0, the VoS is 0. Therefore, the task value can be calculated as follow (Equation 2):

$$\big(Task_j, t\big) = \gamma^{(Task_j)}(w_p * v_p\big(Task_j, t\big) + w_e * v_e(Task_j, t)). \tag{2}$$

Figure 4 shows an example how the value earned by the system for energy consumption can change over a 24 hour period. In Figure 4, the x-axis represents the time interval when the task is submitted, while the y-axis shows the individual task energy consumption (normalized, i.e. the lowest and highest energy consumption vary between 0-1). By using a time aware value function, we can model the fact that gives higher energy values for energy reduction during peak

hours than the values given for non-peak periods. For example, the value for reducing energy during the peak time (e.g., between 12 PM to 6 PM) is four times more important than reducing the same amount of energy during non-peak period (e.g., 12 AM to 5 AM).

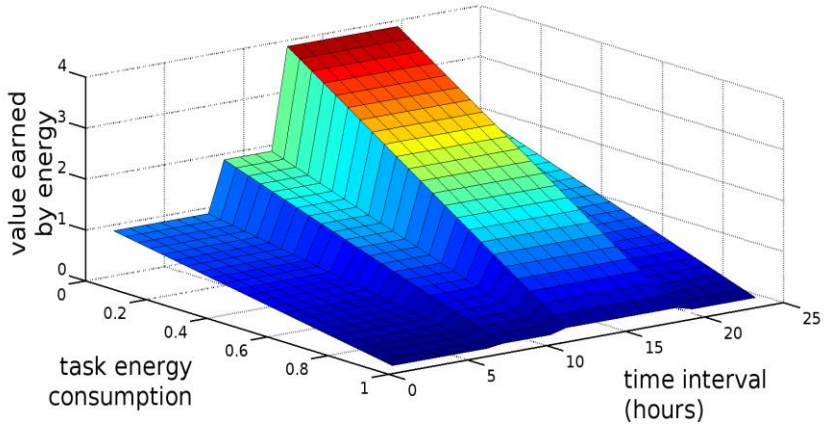


Figure 4. Submission time interval variant energy value.

The VoS can be defined as the total value gained by a workload of n tasks that are completed during a given time interval T as in Equation 3, where t_j represents the submission time of each task executed during the given interval.

$$VoS\ (T) = \sum_{j=1}^{n} V\left(Task_j, t_j\right). \tag{3}$$

2.4. VoS Based Resource Scheduling

2.4.1. Data Driven Energy Modeling

As described in our system model (Section 2.2), each task runs on a separate VM. However, the power/energy consumption of the individual VMs running on a physical machine cannot be measured directly, especially when there are multiple VMs operating concurrently. Therefore, we use a data-driven approach to model the task power consumption. Initially, we collect a wide range

of parameters associated with CPU and memory operations that are the primary contributors to the overall VM power consumption as shown in [22, 23]. We use the Perf tool for tracking hardware performance counters for VMs launched on a hypervisor [24]. First we start with a large set of possible profiling attributes to model the power consumption; however, not all the attributes are highly correlated with the power consumption (An example of a highly correlated attribute is number of cycles. Page faults and context switches have relatively less effect on the power consumption.). Hence to identify the set of performance counters that are highly correlated with power consumption, we generate two sets of micro benchmarks: The first set involves integer and floating point arithmetic operations, such as summation, multiplication, and division, for identifying only CPU intensive parameters. During the arithmetic operations, the memory operations are kept to a minimum. The second set involves memory read and write operations over various sizes of memory blocks. We collect performance counters and power consumption of the VM based on these two sets of micro benchmarks. Then we use our information theory based approach to identify the most relevant parameters for characterizing the power consumption [25]. Table 2 lists the parameters selected for modeling and predicting power consumption. For the CPU attributes, we have observed that the number of cycles, L1 data and instruction cache operations (loads, stores, and misses), data translation lookaside buffer (dTLB) operations (store, load, and store misses), and branch loads have the highest correlation with the power consumption compared to the other attributes. Similarly, for the memory attributes, we have observed that cache misses, CPU and task clock, L1 data cache store and prefetches, instruction loads, last level cache (LLC) operations (load, store, store misses), and dTLB stores with branch loads have higher correlation than other memory attributes.

Table 2. Selected value function parameters.

CPU only filtered parameter set	cycles, L1-dcache-loads, L1-dcache-stores, L1-dcache-store-misses, L1-icache-loads, L1-icache-load-misses, dTLB-loads, dTLB-stores, dTLB-store-misses, branch-loads, Power
memory only filtered parameter set	cache-misses, cpu-clock (msec), task-clock (msec), L1-dcache-stores, L1-dcache-prefetches, L1-icache-loads, LLC-load-misses, LLC-stores, LLC-store-misses, dTLB-stores, branch-loads, Power

In our task power modeling running on the VMs, we mainly focus on the profiling of the allocated VM resources rather than checking the resource utilization percentages. Contrarily, the power models introduced in [22, 23] use resource utilization percentages which is a high level approach; unfortunately, such methods cannot capture the actual operations (number of cycles executed, the cache operations, branch instructions, just to name a few) and result in a high error rate. Therefore, we utilize performance monitor counters for tracking power consumption of individual VMs by incorporating not only low level cache misses but also loads, stores, and pre-fetches along with branch loads.

Once the features have been selected, we can use them to model the power consumption of individual VMs based on the selected performance parameters. We first discretize power values into multiple power levels of 5 Watts each and in the range from 0 to 100 Watts. Discretization at a 5 Watts level reduces the computation complexity of the training process, with small error in power estimates. Based on the data obtained from the micro benchmarks, we model each power level as a function of the performance parameters using the JRip classifier [26]. JRip is a classifier implementation based on the RIPPER algorithm. The RIPPER algorithm starts with an empty rule set and adds new rules as they satisfy the targeted parameter until no new rule can be added. Figure 5 shows an example of a rule generated by the JRip classifier algorithm.

```
if (($L1_icache_loads <= 1036016) && ($L1_dcache_stores >= 379085720) &&
   ($dTLB_stores <= 379613809) ) {
    $Power_ Level = 8;
}
else if (($L1_dcache_loads <= 759961592) && ($L1_dcache_loads >= 705610449) &&
   ($cycles <= 2657633671) && ($L1_dcache_stores <= 379008640)&&
   ($dTLB_store_misses >= 563)) {
    $Power_ Level = 8;
}
…
else {
    $Power_Level = 9;
}
```

Figure 5. Sample rule for the power level 8 (40 Watts).

In our earlier work on dynamic resource allocation for the individual VMs in cloud computing systems, we used a similar approach for characterizing the workload behavior and resource requirements [27, 28, 29]. In this study, we use the same approach in identifying the task power consumption. We train the model based on micro benchmarks and evaluated its accuracy based on NAS-NPB benchmarks [30]. Our model predicts power consumption of the VMs running these benchmarks with 90% accuracy. We discuss these benchmarks in further details in Section 2.6.

1.4.2. Value per Total Resource

Our resource management heuristic algorithm is based on the allocation choices that provide the highest task value divided by the amount of resources used, to better utilize the resources. We introduce a resource management heuristic called *Maximum Value-per-Total Resource (Maximum VPTR)*. The framework for this greedy heuristic is based on the concept of the Min-Min technique [31, 32, 33], but with a very different set of conditions. The Maximum VPTR heuristic selects the choice of resources based on maximizing "task value earned / total amount of resources allocated." We define the total resources for task i as:

$$\begin{aligned} total\ resources\ =\ & execution\ time\ duration\ * \\ & [(\%\ of\ total\ number\ of\ system\ cores\ used) + \\ & (\%\ of\ amount\ of\ total\ system\ RAM\ used)]. \end{aligned} \tag{4}$$

In general, each of these percentages can be weighted for a given system, depending on factors such as the relative cost of cores and memory or relative demand for cores versus memory among applications in typical workloads. For this study, we assumed an equal weighting.

Our heuristic (shown in Algorithm 1) operates as follows. Each task has an associated set of allowable VM configurations (number of cores and amount of memory). Specifically, for each task type, there is a set of *allowable resource configurations*, which are resource configurations for which it is possible for tasks of that type to achieve non-zero value if their execution could start immediately. We define a mapping event as the time when resource management decisions are made; here we assume that mapping event occurs in one-minute intervals. All tasks that have arrived in the system by the time of the mapping event, have not started execution yet, and have not been dropped, are considered as mappable tasks. For each possible mappable task (a task that can earn non-zero value), first, all allowable VM configurations are checked to find out the earliest time these resources (number of cores and the memory amount specified for the allowable VM

configuration) are available. Then, based on the historical completion time and the historical energy consumption, the VPTR for each allowable configuration for that task is calculated, and the VM configuration that provides the highest task VPTR is selected. If there is no configuration for a mappable task that results in a non-zero value, that task is dropped as it cannot contribute any value to the VoS measure. Then, among all mappable task and VM pairs, the pair that has the maximum VPTR is selected and assigned. If there are multiple task and VM pairs resulting in the highest task VPTR, the first available mappable task and VM pair is chosen.

For the case that the VM resource is not immediately available for the mappable task, an allocation in the future needs to be done; hence, we create a place-holder for such tasks. Place-holders are temporary reservations introduced in our earlier work [14] to better allocate resources compared to permanent reservations for dynamically arriving tasks.

The system state information is updated based on the assignment of the task and VM pair or created place-holder, and the task is removed from the list of mappable tasks. This process is then repeated from the beginning until no more mappable tasks exist. Place-holders are removed at the beginning of the next mapping event so they do not block a new task that may provide higher value. It is possible that a task that had a place-holder removed may be reassigned the same resources in the next mapping event. For example, if the task with the first place-holder is the first choice in the next mapping event, it is reassigned those resources.

If there are multiple choices for the VM configurations that have the earliest start time for a given task, we select the cores with the *VM selection procedure* based on our prior work in [14]. Given the cores are homogeneous, a task's execution time will be the same irrespective of which cores are assigned. Because of this, finding the earliest possible completion time for a task (which maximizes the performance value) is equivalent to finding the task's earliest possible start time.

When the earliest possible starting time for a task is found, it is possible that there will be multiple sets of cores that can be used. We use two criteria to choose a set to use. The first of these criteria is to pick cores that cause the smallest number of idle voids in the system (i.e., sections of idle time between the executions of two tasks). The second criterion is to compare the size of the idle voids into which the task would be inserted, and to choose the nodes with the smallest voids. Please see [14] for details. The motivation for these criteria is to reduce the overall fragmentation of the schedule to give future tasks a better chance of being backfilled [14].

Algorithm 1. Pseudo-code for the Max VPTR heuristic.

```
1.  while the set of mappable tasks is not empty
2.      for each task in the set of mappable tasks
3.          find the allowable VM configuration maximizing task VPTR
4.      select task/VM pair that gives the highest VPTR
5.      if selected task can start execution immediately
6.      then
7.          assign selected task to VMs
8.      else
9.          create a place-holder for selected task using its resource allocation choice
10.     remove selected task from mappable tasks
11. end while
```

In our work, the VM resource allocation heuristic considers task arrival time (i.e., submission time). During a day, Figure 6 shows the energy consumption distribution of a general cloud computing system [34]. An increase in energy consumption is seen (which is the typical energy consumption throughout the world [34]) until somewhere between noon and afternoon. Then, the peak energy consumption is observed Energy consumption starts decaying as a working day comes to an end. The energy consumption is at its lowest level around midnight and starts increasing again early in the morning. The energy curve can be directly correlated with the arrival rate of tasks over a period of 24 hours. We use Figure 6 in our design of Table 3, discussed below, to emulate a real life cloud computing environment.

There are multiple ways to capture the arrival time interval impact on value functions. One way is to assign a task type different energy value functions, as shown in Figure 3. Another way

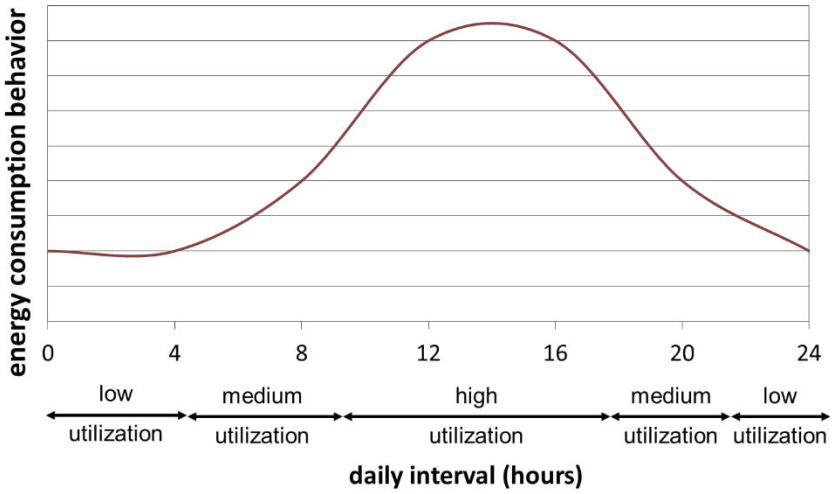


Figure 6. Daily distribution behavior of energy consumption (higher means more energy consumption).

is to use Table 3 to adjust the relative weights of performance and energy in the VoS equation depending on the execution characteristics of a task type and the time of day it was submitted. In our simulation studies and experiments, we use the Table 3 approach.

Our Maximum VPTR (Max VPTR) heuristic uses the performance energy tuples (listed in Table 3) in scheduling its decisions. In our simulations and experiments, tasks are divided into three categories: (1) regular tasks, (2) tasks with a high performance requirement, and (3) tasks that are energy conservative. Furthermore, the arrival time of a task is divided into three categories: low (midnight), medium (morning and evening), and high (around noon) as shown in Figure 6. For a task, the performance and energy weights (w_p, w_e) are shown using the <performance, energy> format in Table 3. During the period when the system resource usage is low (e.g. midnight), both the maximum performance and maximum energy value weights are the

same ($w_p = w_e = 1$). However, during the peak period, the resources are highly utilized and energy consumption reduction is highly desired for large scale computing systems (such as cloud computing systems) for providers to reduce the operational costs hence, the <1, 4> weighting is used for regular tasks. In addition, some tasks with high performance requirements (e.g., disaster recovery applications such as image processing of the environment after an earthquake) are heavily weighted towards performance because energy consumption cost is not as important for these tasks (a weighting of <10, 1> for tasks with high performance requirements in Table 3). In contrast, for tasks that do not require high performance (e.g., creating a report during afterhours), the energy cost reduction would play a more important role regardless of the time the task was submitted (the <10, 1> weightings in Table 3). Even though we have chosen the given weights in Table 3, the constants in the decision making can be adapted for the requirements of a specific system.

In our simulations, we considered comparing against four variants of the Max VPTR heuristic. These heuristics are: (a) Max Value, which prioritizes tasks based on the value that they earn; (b) Max Value-per-Time, which prioritizes tasks based on their value divided by their execution time; (c) Max Value-per-Compute-Resource, which prioritizes tasks based on their value divided by the product of their execution time and number of system cores they use; and (d) Max Value-per-Memory-Resource, which prioritizes tasks based on their value divided by the product of their execution time and the amount of memory they use. Because Max VPTR prioritizes tasks based on their value divided by the total amount of resources that are used (including execution time, number of cores, and amount of memory as defined in Equation 4), it uses all information that is considered by any one of the variations listed above. For this reason, it performed better than or comparable to all of the above heuristics in every scenario we examined. Thus, results with these four variants of Max VPTR are not shown in this paper.

Table 3. Performance and energy decision table fordifferent system usage time and tasks (<performance, energy>).

task\system usage	low	medium	high
regular task	<1, 1>	<1, 2>	<1, 4>
task with high perf. req.	<10,1>	<10,1>	<10,1>
energy conservative task	<1, 10>	<1, 10>	<1, 10>

For comparison to the Max VPTR heuristic, we considered the Simple heuristic, which assigns the mappable tasks in a random order. For each task, it chooses randomly one of its allowable VM configurations, and assigns the task to the earliest possible time. If a task is unable to start executing immediately, then a permanent reservation is created for it on those resources instead [14]. If there are multiple options for the set of cores that have the earliest start time for a given task, Simple just uses the cores with the smallest "id" numbers rather than using the core selection procedure above for Max VPTR. The task is then removed from the set of mappable tasks. This process, shown in Algorithm 2, is repeated until there are no more mappable tasks. We designed, implemented, evaluated, and compared three additional variations of this heuristic. The first, *Simple w/ dropping*, does not make assignments that would earn zero value (it drops the task instead). *Simple w/ place-holders* uses place-holders instead of permanent reservations. Finally, *Simple w/ dropping and place-holders* includes the concepts of place-holders and dropping. Having these three variations shows the benefits of dropping and place-holders.

Algorithm 2. Pseudo-code for the Simple heuristic.

```
1.  while the set of mappable tasks is not empty
2.      select a random task in the set of mappable tasks
3.      randomly choose a VM configuration choice for selected task
4.      if selected task can start execution immediately with that allocation choice
5.      then
6.          assign selected task to that allocation choice
7.      else
8.          create a reservation for selected task using that allocation choice
9.      remove task from mappable tasks
10. end while
```

2.5. Simulation Results

2.5.1. Simulation Setup

We simulate 26 hours of task arrivals and designate the first two hours as warm-up period. Statistics are collected starting from the third hour until the end (24 hours in total). If a task begins execution but does not complete during that interval or finishes execution during the interval but starts its execution prior to the beginning of the interval, then the system will earn a prorated of the task's value. We model ten different system environments. In five, the system memory is fixed at 256 GB and the number of cores is varied between 128 and 384. In the other five, the number of cores is fixed at 256 and system memory is varied from 128 to 384 GB.

We simulate 48 different scenarios with by varying number of tasks, the tasks' type, and the tasks type characteristics (i.e., the allowable resource configurations and associated execution times and energy needs). The number of tasks arriving for each simulation scenario is varied between 1,300 and 1,500 tasks, where each task belongs to one of 30 to 40 task types. The range used for the number of tasks was selected to ensure that the system is oversubscribed. The number of tasks and task types are determined by a uniform distribution and task inter-arrival follows a sinusoidal pattern. The task types are uniformly distributed as regular tasks, tasks with high

performance requirements, and energy conservative tasks as shown in Table 3. Each task has its own performance and energy value functions. The performance and energy value of each task is also weighted based on Table 3. In our simulations, we define system usage for the decision table based on the daily distribution of energy consumption shown in Figure 6. The low usage period is from midnight until 4:00 AM. The medium usage periods are from 4:00 AM until 10:00 AM and from 6:00 PM until midnight. We define the high usage period as 10:00 AM until 6:00 PM.

The method to determine execution times for task types for the simulation studies were based on the COV method used in [35]. Each task type has a number of allowable resource configurations specifying from 4 to 32 cores and from 6 to 40 GB of memory. We create a base case for each task type that defines its execution characteristics given a single core and one GB of memory. The system base case values for a scenario are sampled using Gaussian distributions with means of 150 minutes for execution time and 100 joules for energy consumption. The COV (coefficient of variation) of both distributions is 0.1. For each task type, the base case values are generated with Gaussian distributions that use the system base case value as the mean and a COV of 0.2. If a sampled value falls outside of 40% to 160% of the system base case value, then it is set to the minimum or maximum of this range, respectively. The execution times of the allowable resource configurations are scaled from these base case values of that task type by using the Downey model for the speedup of parallel programs [36], with a uniformly sampled Downey sigma value between 4 and 10. In addition, we scale the resulting execution time based on the amount of memory that is specified in the resource configuration. The energy consumption that corresponds to each resource configuration for a task type is linearly scaled using the number of cores and memory allocated.

Each task type has an importance (γ) between 1 and 10 that is sampled from a Gaussian distribution with a mean that is correlated directly with the task type's base case single core execution time and a COV of 0.2. Each value function has a $v_{max}(Task_j, t)$ value of 1 and a $v_{min}(Task_j, t)$ value distributed uniformly over the range 0 to 1. The hard threshold for the performance value function is created using a Gaussian distribution with a COV of 0.2 and a mean equal to the maximum possible execution time for any VM configuration of the task. The hard threshold for the energy value is generated in a similar way, where the mean is set to the maximum possible energy consumption for any allocation of the task. If the generated value for the hard threshold is less than the corresponding maximum values, the distribution is resampled. The soft thresholds have a COV of 0.2 and a mean equal to the minimum execution time over all VM configurations and minimum energy consumption over all configurations, plus 0.05 times the difference between the minimum and maximum values. A soft threshold is resampled if it is greater than the corresponding hard threshold.

2.5.2. Simulation Results

The results in this section were averaged over 64 simulation scenarios and are shown with 95% confidence intervals. To better compare the performance of heuristics, we plot their total value earned as a percentage of the maximum VoS possible for that scenario. We define the maximum VoS as the VoS earned by the system if all tasks started execution when they arrived in the system with (a) the execution time of their fastest VM configuration allocation choice and (b) the energy consumption of their allocation choice with the lowest energy usage. These values usually come from different VM configuration allocation choices, but they are used to ensure that an upper bound on VoS is calculated in all cases.

Figures 7 and 8 show the percentage of maximum VoS earned by each of the heuristics. Figure 7 shows the system memory fixed at 256 GB with the number of cores varied between 128 and 384. Figure 8 shows the system with number of cores fixed at 256 and system memory varied from 128 to 384 GB. The Simple heuristic has very poor performance in all environments, but its performance improves when dropping or place-holders are used with it. This demonstrates the advantages of using dropping and place-holders.

Our value-based Max VPTR heuristic utilizes additional information to choose tasks and resource configurations that may result in better system performance. Figures 7 and 8 show that the value-based heuristic is able to earn higher value than the Simple heuristic. When the system has a limited number of cores (i.e., 128) or a limited amount of memory (i.e., 128 GB) available, the heuristic is able to perform well because it takes the execution time of and number of cores or amount of memory allocated to a task into account to choose allocation choices that utilize the limited resources (cores or memory) efficiently.

These simulations show that using place-holders and dropping allows the Simple and Max VPTR heuristics to perform well in a variety of environments. The Max VPTR heuristic is able to outperform the Simple heuristic using place-holders and dropping in all of the environments that were simulated. The advantage of the Max VPTR heuristic is that it incorporates the resources needed to execute tasks, and in all our studies of oversubscribed systems at least one of the number of nodes or the amount of memory is a limitation on system performance.

In the following section, we validate the simulation results based on experiments carried out on a physical testbed. We choose the Simple with dropping and placeholder as the baseline heuristic and evaluate the relative performance of the Max VPTR approach for various workload scenarios.

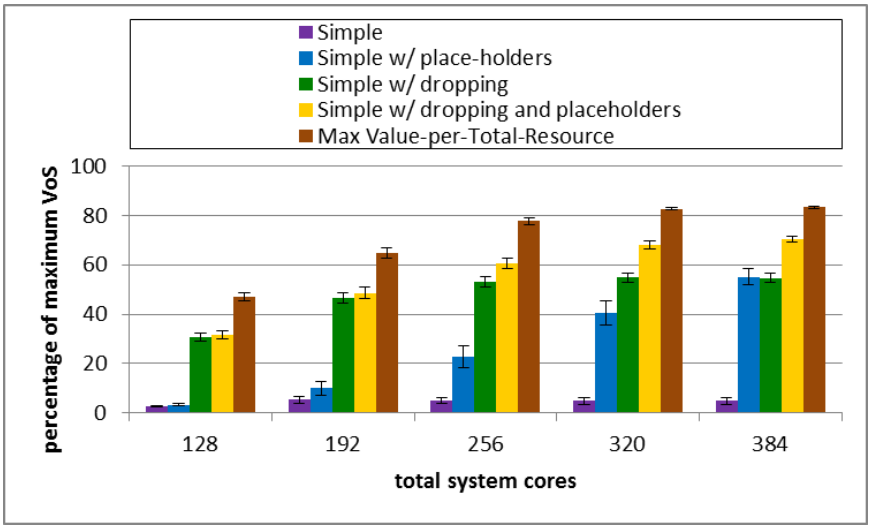


Figure 7. The percentage of maximum VoS earned by the heuristics in environments where the number of cores in the system is varied from 128 to 384 and the amount of memory is fixed at 256 GB.

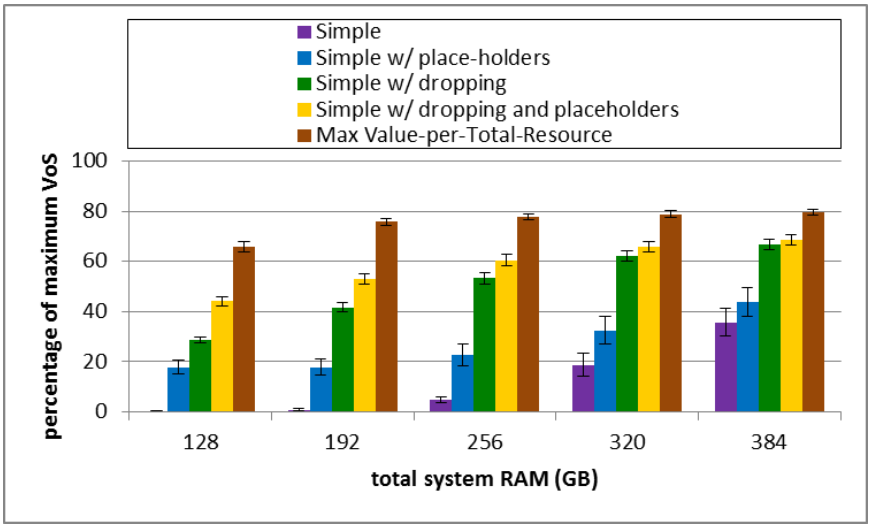


Figure 8. The percentage of maximum VoS earned by the heuristics in environments where the amount of memory in the system is varied from 128 to 384 GB and the number of cores is fixed at 256.

2.6. Experimental Results

2.6.1. Experimental Testbed

We verify the simulation results with experimental results on a real testbed. We use an IBM HS22 blade server with four nodes. Each node consists of two Intel Xeon X5650 6-core processors (2.67 GHz), with 24 threads (two threads per core) and 24 GB RAM. We treat each of the 24 threads as an individual CPU and allocate 20 of them for VMs and the remaining threads are left for the services, programs, and local controllers running on the physical machines. Each node runs the Kernel-based Virtual Machine (KVM) hypervisor [37] and the Perf tool [24]. We have chosen KVM and Perf tool because Perf tool is able to profile KVM based VM resources [24]. We record the power consumption of each node using IBM Advanced Management Module [38].

A VM is assigned to a specific number of cores and amount of memory on a single node in our blade server (i.e., allowable VM configurations). A VM cannot be split across multiple nodes, but a single node can have multiple VMs. Each task is assigned to a specific VM and a task cannot be split across multiple VMs. Resource allocation for a VM on a particular node is limited by the node's physical resources available at that point of time and VMs do not share hardware resources. Hence, on a node, if two out of four cores are not assigned to any VM during resource allocation, then those two idle cores can be assigned to another VM only if the core count requirement is less than or equal to two for a new task.

2.6.2. Workload and Generation

We model our workload as a set of independent tasks that arrive dynamically. To represent such a workload scenario, we use the following four kernels from the MPI based NAS-NPB [30] benchmarks where each kernel is a task type as follows:

- FT - discrete 3D fast Fourier Transform
- IS - Integer Sort, random memory access
- EP - Embarrassingly Parallel
- LU - Lower-Upper Gauss-Seidel solver

This benchmark suite allows users to adjust the problem size, ranging from class A (smallest) to F. For example, for a FT kernel, class A processes a 256x256x128 matrix, whereas class F processes a 4096x2048x2048 matrix. For the experiments, we use class C which processes a matrix of 512x512x512 for FT kernel because of its data size and computational requirements.

We generate a workload trace for simulating the task arrival and store it in a queue that is being monitored by the scheduler. Each generated task is associated with a task type (i.e., NAS-NPB kernel type), its arrival time, and soft and hard thresholds for completion time to represent the performance requirement of each task. The task type is chosen using a uniform distribution. During the workload trace generation, we used similar method to the simulation environment. For the soft completion time thresholds of the submitted tasks, we used a COV of 0.2 and mean equal to the minimum execution time over all VM configurations. For the soft energy consumption threshold, we used minimum energy consumption over all configurations, plus 0.05 times the difference between the minimum and maximum values. The hard thresholds are created using Gaussian distributions with a COV of 0.2 and means equal to the highest execution time in the historical data.

The difference between arrival times of two consecutive tasks is sampled from using a Gaussian distribution. The inter-arrival time of tasks are sampled such that the system is oversubscribed for high utilization periods. Inter-arrival rate (number of tasks) for the medium and

low utilization periods have reduced 30% and 60% compared to highly utilized period. Additionally, in the experimental results, we assume that each task has the same importance (γ).

2.6.3. Overall Architecture

Figure 9 illustrates the overall scheduler architecture. Our scheduler design consists of a queue, the resource manager, and the cloud system. When a task is submitted by the user, the following information is provided: task type, arrival time (i.e., when the task is submitted to the system), and its completion time thresholds information. The resource manager applies the allocation heuristic. Based on the user submitted information and the information generated by the resource allocation heuristic, tasks are stored in the queue with the following information regarding each task: TaskID (i.e., task sequence number), task type, arrival time, soft and hard thresholds (please note that while the user is interested in the execution time, the service provider is interested mainly in energy consumption to reduce the operational cost and hence the energy thresholds are imposed by the service provider), and task execution information (e.g., which host it will be running on and with what VM resources, when its execution starts and ends, and if the task has been scheduled or not).

Each heuristic under investigation uses the Estimated Time to Compute (ETC) matrix and the Estimated Energy Consumption (EEC) matrix to determine VM allocations and scheduling. From historical data, we create the ETC matrix, where $ETC(i,j)$ is the estimated time to compute a task of type i on a VM configuration of type j. Similarly, we create the EEC matrix, where $EEC(i,j)$ is the estimated amount of energy consumed during execution by task of type i on a VM configuration of type j (i.e., amount of allocated resources in terms of the number of cores and amount of memory) when no other VM is scheduled on the cluster. Using historical data is a common approach in large scale computing scheduler studies as shown in [10, 11]. Please note that to reduce the complexity of our model the entries in the ETC and EEC matrices include the

overhead time and energy for spawning a VM to run the task. In our environment, the number of cores that can be allocated for a VM is $\#cores = \{1, 4, 8, 16\}$ and the amount of RAM that can be allocated is $memory = \{1, 2, 4, 8, 16\}GB$.

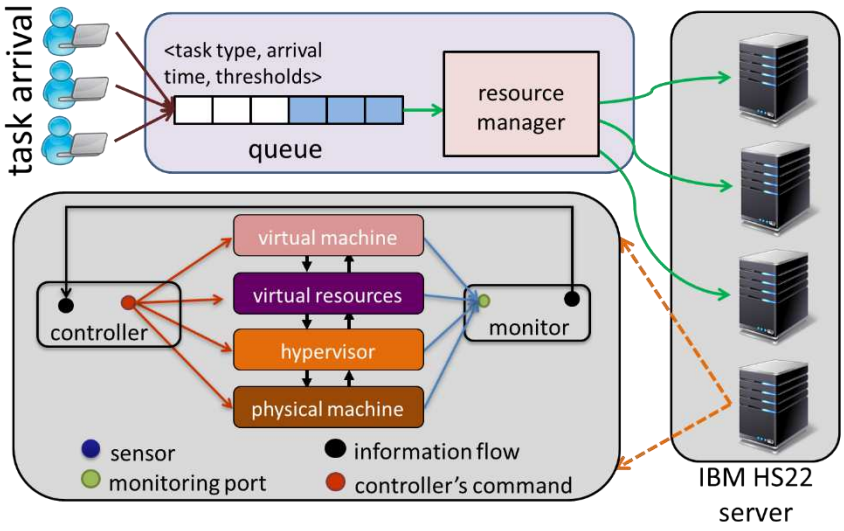


Figure 9. Overall scheduler architecture.

Depending on the resource manager assignments, a local controller, which resides on each node, generates the required VM, starts the Perf tool on the physical host machine (to profile the VM resource usage), and executes its assigned task on the VM. The algorithm of the local controller is shown in Algorithm 3. To avoid VM generation overhead, we generated a set of VMs ready to be launched with the required executables. The function *create_VM* allocates the assigned resources for the VM, if these resources are available. Upon successful VM start, the IP of the VM is retrieved using the Address Resolution Procotol (ARP) cache [55, 56]. When the task is completed, the value of all the monitored performance counters along with task execution time are used to calculate energy consumption using the power model introduced in Section 2.4.1. After

the termination of the VM, the local controller updates the task queue indicating the completion of task execution.

Algorithm 3. Pseudo-code for the local controller

```
1.    local_controller (input: task, #core, #memory)
2.    VM_IP = create_VM (#core, #memory)
3.    if(!VM_IP)
4.        update queue (task cannot be run)
5.    execute Perf on host for task
6.    VM_execute(VM_IP, task)
7.    stop Perf
8.    kill VM

1.    function create_VM (input: #core, #memory)
2.      check if #core & #memory available
3.      $VM = find_idle_VMs(template VMs);
4.      configure_VM($VM, #core, #memory)
5.      create_VM($VM)
6.      $VM_IP =get _IP (VM)
7.      return VM_IP
```

We calculate the value earned by the system after completing each task using Equation 2. The earned values for all the tasks are accumulated into workload's VoS at the end of the experiment. We use the VoS metric for comparing the performance of the heuristics for each workload type (trace).

2.6.4. Experimental Results

In this section, we evaluate our Max VPTR approach and compare its performance with respect to the Simple heuristic with dropping and placeholder (shown as Simple w/ d&p) in terms of the tasks' total execution time, total energy consumption, and total system VoS earned. We have used two workload cases for our evaluations. The first workload is six hours long and it mimics the daily utilization behavior of the large-scale systems, shown in Figure 6, such a way that the tasks' arrival time changes in the following order: low utilization, medium utilization, high utilization,

high utilization, medium utilization, and then low utilization, creating a complete cycle. For the second workload, we assume that the system resource usage is always at its peak (high utilization) during its execution of three hours. We evaluate and compare the task scheduling algorithms for the first workload using results shown in Figures 10 through 14 and for the second workload using results shown in Figures 15 through 17. In our experimental evaluation, we varied the total number of cores available to be 20, 40, and 80 and varied the number of tasks proportionally. Please note that for the execution time of individual task, we use the difference of the task execution time and completion time and for the energy consumption of a task, we use the VM resource usage and the power model discussed in Section 2.4.1. In addition, for workload 1, the weights of the performance and energy values (w_p and w_e in Equation 2), are chosen based on the values shown in Table 3.

Figure 10 shows the total task execution time (sum of the execution times measured upon completion, for all tasks) for the Max VPTR and Simple w/ d&p methods. Figure 11 shows the total energy consumption for all the tasks associated with workload 1. The total execution time and the total energy consumption are the accumulation of the measured results for the scheduled tasks that complete. Hence, as we increase the number of cores, we observe that the total execution time increases for both heuristics because the number of tasks arriving to the system and complete within the six hour time window increases with more cores.

In these experiments, we can see that the Max VPTR approach performs better than the Simple w/ d&p algorithm with respect to both performance (i.e., total execution time) and total energy consumption. This is because the Max VPTR heuristic makes informed decision in resource allocation by taking energy and execution time into account during the resource scheduling. On the other hand, Simple w/ d&p applies a random resource scheduling resulting in higher total

execution time and energy consumption. On the other hand, Simple w/ d&p algorithm randomly maps the VM resources as long as the hard thresholds for performance and energy are met.

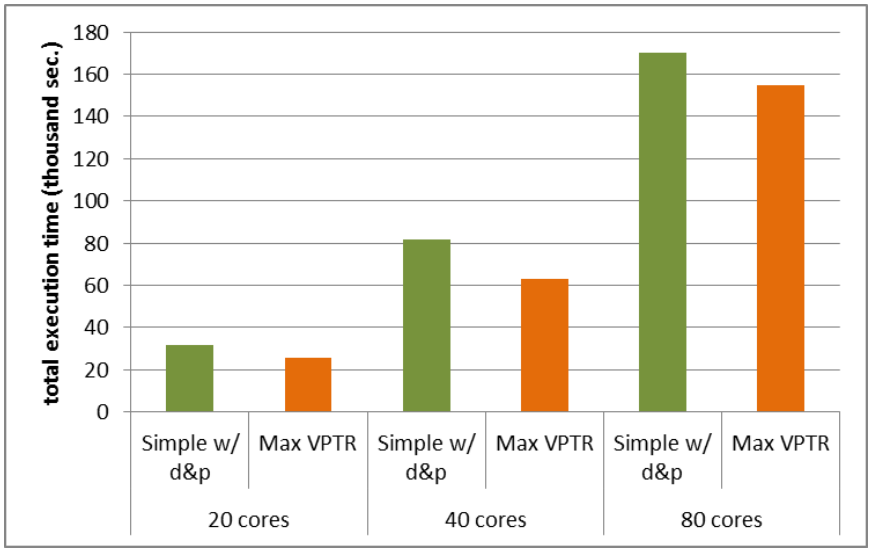


Figure 10. Total task execution time for workload 1 (thousand seconds).

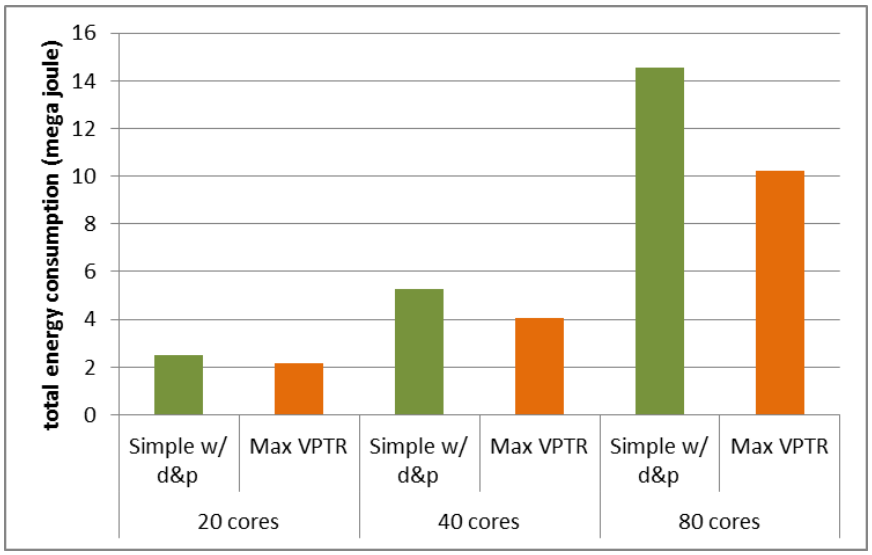


Figure 11. Total energy consumption by the executed tasks for workload 1 (in mega joules).

In Figure 12 and Figure 13, we show the total performance value and total energy value earned, respectively, by both heuristics. Total performance value is the sum of the performance values of

all tasks completed. Total energy value is defined similarly. For the comparison, we plot the total value earned as a percentage of the maximum possible VoS that can be earned if all the submitted tasks complete before their soft completion thresholds and energy thresholds, as in Section 2.5.2. The experimental results show that the Max VPTR has better results in terms of performance (up to 82% improvement in the best case) and energy (up to 110% improvement in the best case) values gained for the workload 1 when compared to the Simple w/ d&p heuristic method.

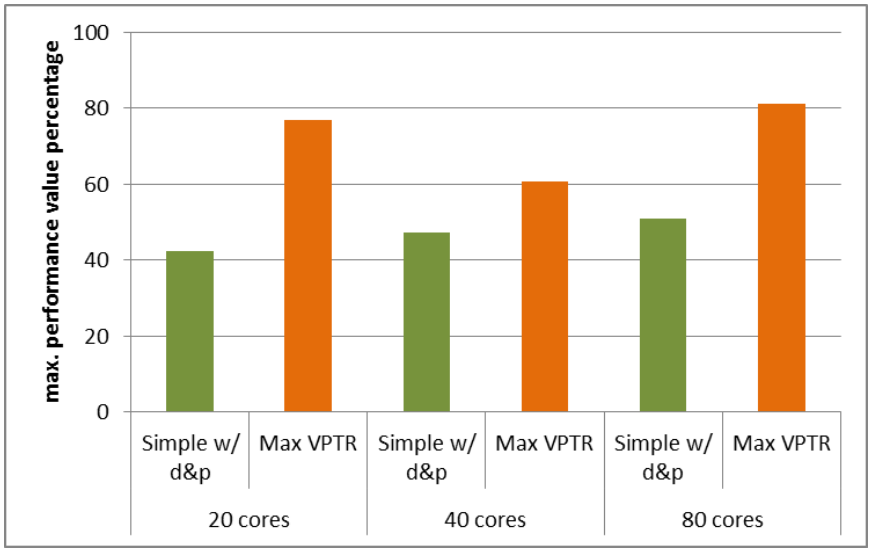


Figure 12. The percentage of maximum performance value earned by the heuristics for workload 1.

In Figure 14, we show the percentage of the maximum VoS obtained by both heuristics for workload 1. For the VoS calculation, Equation 3 is used and the percentage of the maximum theoretical earnable VoS is calculated as was discussed in Section 2.5.2. The results show that the VPTR method achieves better results consistently by providing up to 77% more value when 80 cores are used, 36% for 40 cores, and 32% for 20 cores. During our experiments, we have observed that both Simple w/ d&p and Max VPTR were able to show a close behavior in terms of energy consumption due to the limited available resources resulting in similar resource mappings. While

the available resources increase (80 cores, as an example), Max VPTR algorithm can make more effective resource mapping for tasks.

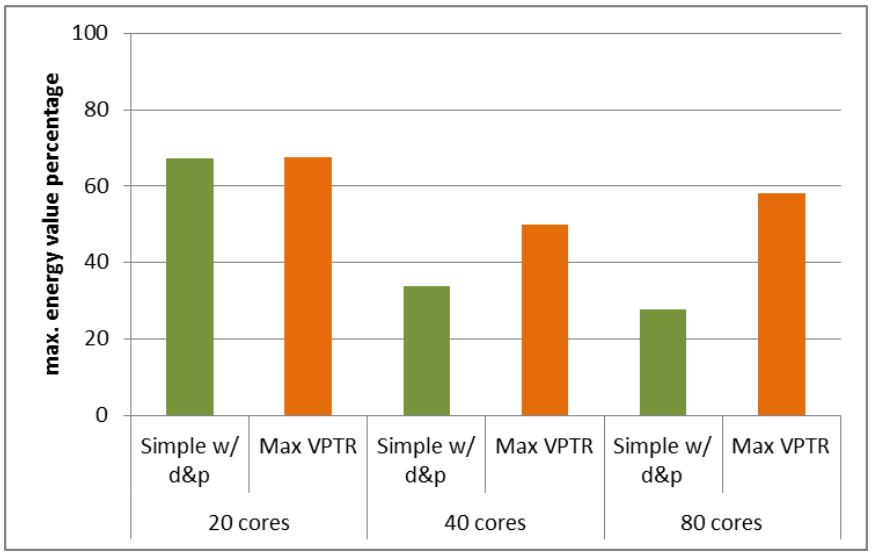


Figure 13. The percentage of maximum energy value earned by the heuristics for workload 1.

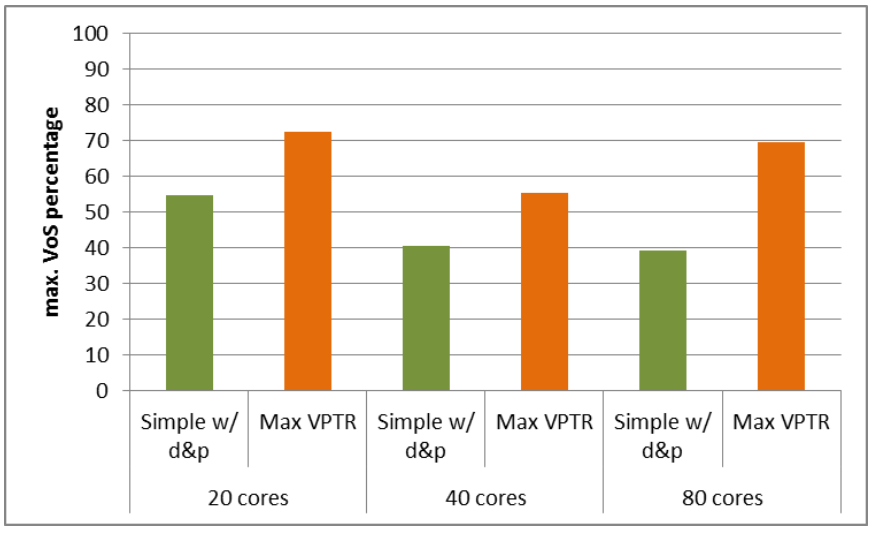


Figure 14. The percentage of maximum VoS earned by the heuristics for workload 1.

Our observations on the experimental results are in agreement with the simulation results shown in Section 2.5. The Max VPTR method outperforms the Simple w/ d&p in both simulation and experimental results. In Figure 7, we observe that the Max VPTR results are up to 49% higher than the Simple w/ d&p (for the best case). However, we observe a higher difference in the experimental results. Because each physical system is limited to 20 cores for the VM environment, the Max VPTR algorithm outperforms the Simple w/ d&p algorithm by managing the resources more effectively and this difference can be better seen when there are more tasks submitted to the system.

On a real testbed, the completion time and energy consumption of a task can vary because of overall system usage from other tasks, their interference to the other tasks (because the physical hardware is shared among all the VMs running tasks), services running in the background, and so on. Therefore, we observe differences between the historical data (ETC and EEC matrices) used by the heuristics in the experiments and the actual measured performance and energy consumption values. Furthermore, the simulation studies use execution characteristics that are independent of the overall system usage. This is one of the root causes for observing variations in percentage difference between the two heuristics over simulation and experimental results. In addition, there are the differences between the simulation and the experimental environments such as the task inter-arrival time, the allowable resource configurations, and the use of variations in the importance of the tasks (γ) in the simulations but not the experiments.

For the second workload type (where we assume we are always at high system usage), we have allocated different number of cores and adjusted the task inter-arrival times. In Figures 15 and 16, we present the performance and energy value percentages earned by both Max VPTR and Simple w/ d&p heuristics. The experimental results show similar behavior with the experimental results

of workload 1 with respect to Max VPTR getting higher percentages than Simple w/ d&p. For the configuration with 80 cores, Max VPTR achieved an increase of almost 100% in performance value and 84% in energy value.

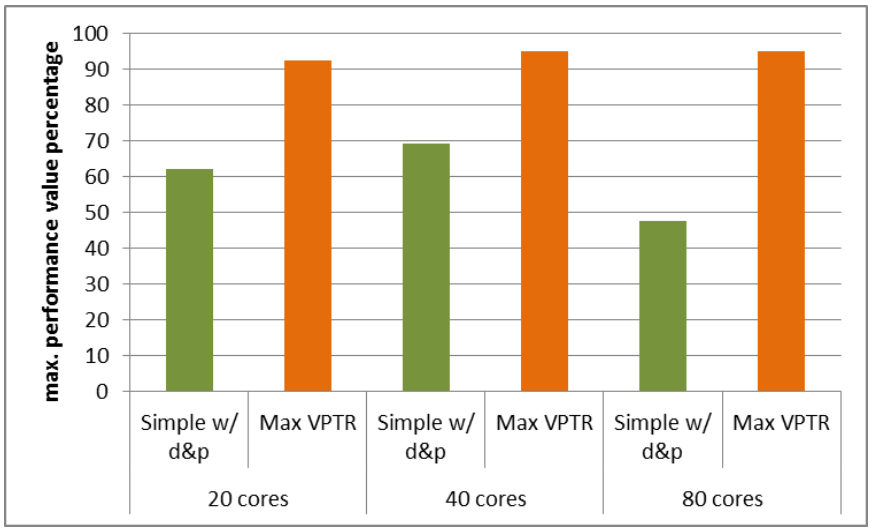


Figure 15. The percentage of maximum performance value earned by the heuristics for workload 2.

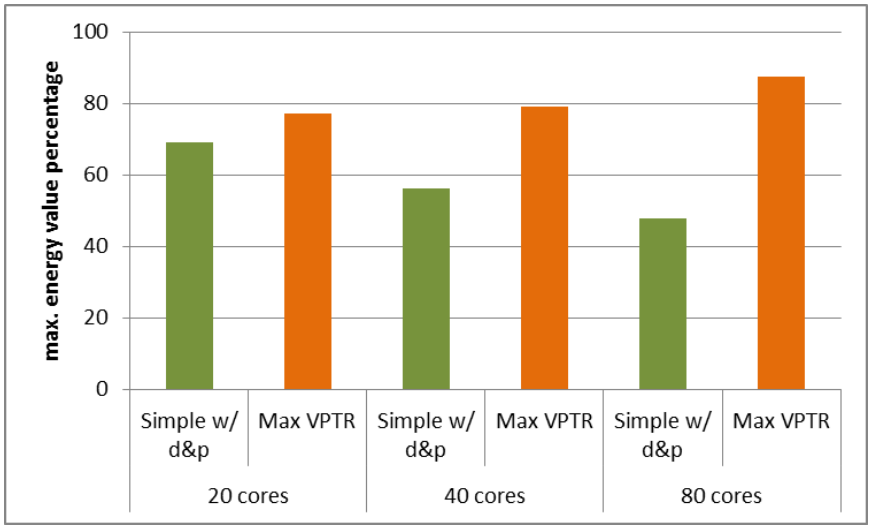


Figure 16. The percentage of maximum energy value earned by the heuristics for workload 2.

Similar to Figure 14, we have calculated the VoS for workload 2 (Figure 17). The results show similar behavior to the workload 1 with Max VPTR outperforming the Simple w/ d&p heuristic

method. In this case, however, we see a higher gap between the values of the two heuristics and the gap increases as we increase the resources. Because workload 2 involves high system usage, the Simple w/ d&p heuristic is not able to make efficient decisions in utilizing the resources, resulting Max VPTR having up to 91% higher VoS. Due to the randomness of the Simple w/ d&p heuristic when considering the amount of resources to be allocated for each task (i.e., the VM configurations), while the number of cores increases, the percent of maximum VoS reduces. In contrast, the Max VPTR heuristic utilizes the resources effectively considering the limited available resources.

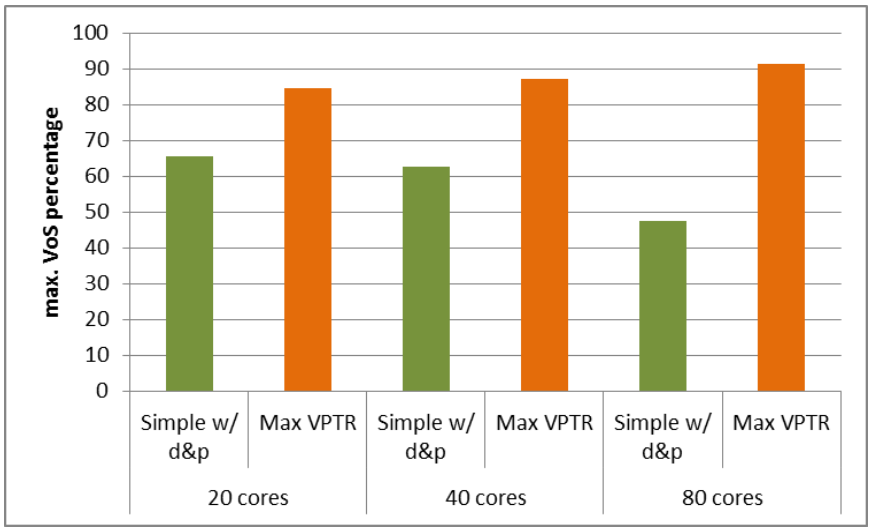


Figure 17**. The percentage of maximum VoS earned by the heuristics for workload 2.**

2.7. Related Work

There is a large body of work on resource management strategies for cloud computing systems targeting various metrics such as system resource utilization, task execution time, total power/energy consumption, and system resources. Most of the resource management algorithms for parallel tasks utilized the backfilling ([14, 39-45]) approach for improving resource utilization and average execution time. Utility functions have been shown to be an effective performance

measure for resource management in an oversubscribed environment. However, utility functions have been mainly defined only as time varying utility of completing a task for the user. “Value function” based methods ([45-48]) that rely on monotonically decreasing value measurements are similar to the utility based methods for resource management. A primary difference of our work from these studies is the fact that the VoS metric considers a weighted combination of the value of energy reduction and the task completion time. Furthermore, the weights for performance versus energy are a function of category of task type and the time of day task is submitted (Table 3). As discussed before, the value of reducing energy at peak consumption has much higher value than the period for low energy consumption.

For example, the value function introduced in [14] quantified the useful work done in an oversubscribed system in terms of total value accumulated by completing tasks in a timely fashion. The new heuristics proposed in [14] outperform the Conservative Backfilling and EASY (Extensible Argonne Scheduling sYstem) Backfilling algorithms. However, the value function took only task completion time into account. Furthermore, its system model did not consider VMs and the impact of configurations that consist of number of cores and amount of memory allocated.

The utility based heuristic introduced in [16] allowed scheduling only the tasks that did not violate the allotted energy budget within a given time period. Utility maximization was also studied in [17] under an energy constraint during oversubscription by prioritizing the scheduling of tasks with the highest magnitude of utility per unit energy at a particular instant of time. These techniques did not consider the time at which the resource scheduling is performed (during peak or non-peak periods), and only considered an energy constraint rather than an energy value. Furthermore, these works did not consider VMs and the impact of configurations that consist of number of cores and amount of memory allocated.

A power aware job scheduling approach was presented in [49] for HPC systems by treating scheduling as a 0/1 knapsack problem. Jobs were scheduled to maximize the resource utilization without exceeding a given power budget. However, this method did not consider the importance of total energy consumption of a task, as well as not considering the time of day a task is submitted. Their system model also did not consider using VMs for tasks.

In [50], task scheduling has been applied to reduce the power consumption while considering the system thermal state, fan speeds, and network operations. They first determined that the load across a data center was not fairly distributed causing some CPUs highly loaded and some almost idle. This resulted in an imbalance in the power consumption of the other components as well. Hence, the authors suggested that by reducing the imbalance across the power consumption of the system components, it was possible to achieve a 12% reduction in power consumption. However, they did not consider the individual task requirements (i.e., number of cores and memory amount) and focused on balancing the CPU load only. And, they did not consider a virtualized environment as well. Laszewski et al. [51] presented a power aware clustering algorithm where they applied DVFS. But the proposed approach only used DVFS and did not include the core and memory requirements. In [3] Tang et al. proposed an energy aware scheduling algorithm for the optimization of the energy savings using only DVFS technique for the heterogeneous systems. Their approach distributed the parallel applications to meet the required deadline while reducing energy consumption. They demonstrated their approach using CloudSim simulator only. In [52], Ding et al. allocated VMs to physical hosts based on performance-power ratio and their idle cores by finding an optimal frequency to run a VM. They also migrated the active VMs to the physical hosts with higher performance-power ratio to reduce the energy and increase the processing capacity. Their simulation results showed over 20% reduction of energy and 8% increase of

processing capacity in the best cases, compared to the algorithm defined in [51]. Their work can be complementary to our approach for the heterogeneous environment; however, they did not consider multiple task type and their performance and energy requirements with soft and hard thresholds and they only provided simulation based results that simplified many issues such as overhead.

In [53], task scheduling was applied for green data centers where mainly solar energy is used. They predicted both the workload requirements and the amount of available solar energy so that low priority workloads can run with the solar energy and for the workloads that should run with the electrical grid, they try to assign them during the periods when the cost of electricity is less than other periods. This work can be a complementary approach to reduce the operational cost; nevertheless, by applying a finer grain task scheduling approach (e.g., this work), it is possible to further reduce the power consumption resulting in reduction in overall cost.

A profit- and penalty-aware scheduling algorithm introduced in [18] associated a task with two different Time Utility Functions (TUFs): a profit TUF and penalty TUF. The profit TUF was defined as the utility earned by a system based on timely completion of a task. Penalty TUF is defined in terms of utility lost by dropping the task or missing its completion deadline. In our approach, we do not apply penalty utility to the tasks that were not completed. Instead, we schedule a task only if it can earn any value, and once scheduled a task cannot not be dropped. Also, for [18], the earlier task completion provided the highest utility, while our approach includes a relaxation to the problem with soft and hard thresholds for the performance and energy functions. Finally, our approach also considers the energy consumption upon a task completion.

In [54], the authors proposed new heuristics for an energy-constrained environment with parallel tasks. These techniques were shown to outperform existing FCFS-based techniques in the

parallel environment. However, this work optimized for performance while obeying the energy constraint instead of considering a metric that combines performance and energy as we do in this work. In addition, [54] did not model the memory assigned to tasks, evaluated the proposed heuristics through simulations only, and did not verify the simulations through experiments on real systems. Finally, in [54], resources were allocated for the case where each task has a single choice for its resource configuration (i.e., the number of cores) in each heterogeneous cluster. Our work considers multiple resource configurations (i.e., both the number of cores and amount of memory) in a single homogeneous cluster.

2.8. Conclusion

Large scale computing systems and data centers typically follow a resource utilization pattern that is low during times such as midnight and high during the mid-day hours. Therefore the job submission time plays an important role in the resource management for these systems. In this paper, we presented the Value of Service (VoS) metric to measure the total value earned by completing the tasks during a given time interval and also based on the energy consumption of each task. We model the value of energy reduction and performance of task completion for peak and non-peak periods. We use several cases where the system demands change over time and the VoS metric is used to balance the user requirements as well as the cloud service provider objective to reduce operational cost. We show how we can weigh the impact of performance versus energy in calculating the VoS depending on the category of a task (e.g., high performance, energy conservative) and the time of day when it is submitted.

We develop a system model based on VM configurations. Each VM configuration is specified by the number of cores and amount of memory that will be assigned to a task type. We design a set of resource management heuristics that attempt to maximize the time-of-use system VoS

metric. We evaluate and compare these heuristics, first by simulation and then validate the main simulation result with experiments on an IBM HS22 blade server. In particular, we show in our simulations that Max Value-per-Total Resource (Max VPTR) outperforms the Simple with dropping and placeholder algorithm and its variations. We then compare the best Simple variation with Max VPTR with experimentation on an IBM based testbed. Our simulation and experimental results showed the effectiveness of our Max VPTR approach when compared with the Simple with dropping and placeholder heuristic approach.

References

[1] J. Luo, L. Rao, and X. Liu, "eco-IDC: Trade Delay for Energy Cost with Service Delay Guarantee for Internet Data Centers," *IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 45–53, Sep. 2012.

[2] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The Cost of a Cloud: Research Problems in Data Center Networks," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 68–73, Dec. 2008.

[3] Z. Tang, L. Qi, Z. Cheng, K. Li, S. U. Khan, and K. Li, "An Energy-Efficient Task Scheduling Algorithm in DVFS-Enabled Cloud Environment," *Journal of Grid Computing*, vol. 14, no. 1, pp. 55–74, Mar. 2016.

[4] P. Delforege, "Critical Action Needed to Save Money and Cut Pollution," Feb. 2015, ([Online]: https://www.nrdc.org/resources/americas-data-centers-consuming-and-wasting-growing-amounts-energy), accessed May 2016.

[5] P. Delforage and J. Whitney, "Scaling Up Energy Efficiency Across the Data Center Industry: Evaluating Key Drivers and Barriers," Issue Paper, National Research Defnse Council (NRDC), Aug. 2014.

[6] J. Koomey, "Growth in Data Center Electricity Use 2005 to 2010," Analytics Press, Aug. 2011, ([Online]: http://www.analyticspress.com/datacenters.html), accessed May, 2016.

[7] Top Ten Exascale Resource Challenges, Technical Report, DOE ASCAC, Feb. 2014, ([Online]: http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf), accessed Aug. 2015.

[8] L. Rao, X. Liu, L. Xie, and W. Liu, "Minimizing Electricity Cost: Optimization of Distributed Internet Data Centers in a Multi-Electricity-Market Environment," *IEEE INFOCOM*, 9 pp., Mar. 2010.

[9] Amazon ECE2 Reserved Instances, AWS, Amazon, ([Online]: https://aws.amazon.com/ec2/purchasing-options/reserved-instances), accessed May 2016.

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

[11] D. Xu, K. Nahrstedt, and D. Wichadakul, "QoS and Contention-Aware Multi-Resource Reservation," *Cluster Computing*, vol. 4, no. 2, pp. 95–107, Apr. 2001.

[12] E. Caron, F. Desprez, and A. Muresan, "Forecasting for Grid and Cloud Computing On-Demand Resources Based on Pattern Matching," *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom 2010)*, pp. 456–463, Nov. 2010.

[13] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic Resource Allocation and Parallel Execution in the Cloud for Mobile Code Offloading," *2012 IEEE Infocom*, pp. 945–953, Mar. 2012.

[14] B. Khemka, D. Machovec, C. Blandin, H. J. Siegel, S. Hariri, A. Louri, C. Tunc, F. Fargo, and A. A. Maciejewski, "Resource Management in Heterogeneous Parallel

Computing Environments with Soft and Hard Deadlines," *11th Metaheuristics International Conference (MIC 2015)*, 10 pp., June 2015.

[15] B. Khemka, R. Friese, L. D. Briceño, H. J. Siegel, A. A. Maciejewski, G. A. Koenig, C. Groer, G. Okonski, M. M. Hilton, R. Rambharos, and S. Poole, "Utility Functions and Resource Management in an Oversubscribed Heterogeneous Computing Environment," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2394–2407, Aug. 2015.

[16] B. Khemka, R. Friese, S. Pasricha, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, S. Powers, M. Hilton, R. Rambharos, and S. Poole, "Utility Maximizing Dynamic Resource Management in an Oversubscribed Energy-Constrained Heterogeneous Computing System," *Sustainable Computing: Informatics and Systems*, vol. 5, pp. 14–30, Mar. 2015.

[17] H. Wu, B. Ravindran, and E. D. Jensen, "Energy-Efficient, Utility Accrual Real-Time Scheduling Under the Unimodal Arbitrary Arrival Model," *IEEE/ACM Design, Automation and Test in Europe (DATE 2005)*, pp. 474–479, Mar. 2005.

[18] S. Liu, G. Quan, and S. Ren, "On-Line Scheduling of Real-Time Services for Cloud Computing," *6th World Congress Services (SERVICES 2010)*, pp. 459–464, July 2010.

[19] M. Snir and D. A. Bader, "A Framework for Measuring Supercomputer Productivity," *International Journal of High Performance Computing Applications*, vol. 18, no. 4, pp. 417–432, Nov. 2004.

[20] D. Machovec, C. Tunc, N. Kumbhare, B. Khemka, A. Akoglu, S. Hariri, and H. J. Siegel, "Value-Based Resource Management in High-Performance Computing Systems," *7th Workshop on Scientific Cloud Computing (SCIENCECLOUD 2016)*, pp. 19–26, May/June 2016.

[21] C. Tunc, N. Kumbhare, A. Akoglu, S. Hariri, D. Machovec and H. J. Siegel, "Value of Service Based Task Scheduling for Cloud Computing Systems," *IEEE 2016 International Conference on Cloud and Autonomic Computing (ICCAC 2016)*, 11 pp., Sep. 2016.

[22] A. E. Bohra and V. Chaudhary, "VMeter: Power Modelling for Virtualized Clouds," IEEE International Symposium on Parallel & Distributed Processing, *Workshops and Phd Forum (IPDPSW'10)*, 8 pp., April 2010.

[23] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya, "Virtual Machine Power Metering and Provisioning," *1st ACM Symposium on Cloud Computing (SoCC 2010)*, pp. 39–50, June 2010.

[24] "Perf: Linux profiling with performance counters," ([Online]: https://perf.wiki.kernel.org/index.php/Main_Page), accessed Nov. 2015.

[25] G. Qu, S. Hariri, and M. Yousif, "A New Dependency and Correlation Analysis for Features," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 9, pp. 1199–1207, Aug. 2005.

[26] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, June 2009.

[27] F. Fargo, C. Tunc, Y. A. Nashif, and S. Hariri, "Autonomic Performance-per-Watt Management (APM) of Cloud Resources and Services," *ACM Cloud and Autonomic Computing Conference (CAC 2013)*, 10 pp., Aug. 2013.

[28] C. Tunc, “Autonomic Cloud Resource Management,” PhD Thesis, Electrical and Computer Engineering Deaprtment, University of Arizona, Tucson, Arizona, USA, 2015.
[29] F. Fargo, C. Tunc, Y. Al-Nashif, A. Akoglu, and S. Hariri, “Autonomic Workload and Resources Management of Cloud Computing Services,” *International Conference on Cloud and Autonomic Computing (ICCAC 2014)*, pp. 101–110, Sep. 2014.
[30] “NAS Parallel Benchmarks (NAS-NPB),” NASA Advanced Supercomputing, ([Online]: https://www.nas.nasa.gov/publications/npb.html), accessed Aug. 2015.
[31] T. D. Braun, H. J. Siegel, N. Beck, L. Boloni, R. F. Freund, D. Hensgen, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, “A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems,” *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, 2001, pp. 810–837.
[32] O. H. Ibarra and C. E. Kim, “Heuristic Algorithms for Scheduling Independent Tasks on Non-Identical Processors,” *Journal of the ACM (JACM)*, vol. 24, no. 2, pp. 280–289, Apr. 1977.
[33] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, “Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems,” *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 107–131, Nov. 1999.
[34] A. H. Mohsenian-Rad, V. W. S. Wong, J. Jatskevich, and R. Schober, “Optimal and Autonomous Incentive-Based Energy Consumption Scheduling Algorithm for Smart Grid,” *Innovative Smart Grid Technologies (ISGT 2010)*, 6 pp., Jan 2010.
[35] S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, and S. Ali, “Representing Task and Machine Heterogeneities for Heterogeneous Computing Systems,” *Tamkang Journal of Science and Engineering*, Special Tamkang University 50th Anniversary Issue, vol. 3, no. 3, 2000, pp. 195–207.
[36] A. B. Downey, “Model for Speedup of Parallel Programs,” Technical Report UCB/CSD-97-933, Berkeley, 1997.
[37] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “KVM: The Linux Virtual Machine Monitor,” *Linux Symposium*, vol. 15, pp. 225–230, July 2007.
[38] “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 Aug 2015.
[39] D. A. Lifka, “The ANL/IBM SP Scheduling Systems,” *Workshop on Job Scheduling Strategies for Parallel Processing (IPPS 1995)*, pp. 295–303, Apr. 1995.
[40] A. W. Mu’alem and D. Feitelson, “Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 529–543, June 2001.
[41] R. Davis and A. Burns, “A Survey of Hard Real-Time Scheduling for Multiprocessor Systems,” *ACM Computing Surveys*, vol. 43, no. 4, 35 pp., Oct. 2011.
[42] D. Feitelson and L. Rudolph, “Parallel Job Scheduling: Issues and Approaches,” *Workshop on Job Scheduling Strategies for Parallel Processing (IPPS 1995)*, 18 pp., Apr. 1995.

[43] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong, "Theory and Practice in Parallel Job Scheduling," *Workshop on Job Scheduling Strategies for Parallel Processing (IPPS 1997)*, 34 pp., Apr. 1997.

[44] D. G. Feitelson, U. Schwiegelshohn, and L. Rudolph, "Parallel Job Scheduling - A Status Report," *10th International Workshop on Job Scheduling Strategies for Parallel Processing (JSPP 2004)*, Lecture Notes in Computer Science, pp. 1–16, June 2004.

[45] A. Mishra, S. Mishra, and D. S. Kushwaha, "An Improved Backfilling Algorithm: SJF-B," *International Journal on Recent Trends in Engineering & Technology*, vol. 5, no. 1, pp. 78–81, Mar. 2011.

[46] E. Jensen, C. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Systems," *IEEE Real-Time Systems Symposium*, pp. 112–122, Dec. 1985.

[47] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Realtime Operating Systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 613–629, Sep. 2004.

[48] D. Vengerov, L. Mastroleon, D. Murphy, and N. Bambos, "Adaptive Data-Aware Utility-Based Scheduling in Resource-Constrained Systems," *Journal of Parallel and Distributed Computing*, vol. 70, no. 9, pp. 871–879, Sep. 2010.

[49] Z. Zhou, Z. Lan, W. Tang, and N. Desai, "Reducing Energy Costs for IBM Blue Gene/P via Power-Aware Job Scheduling," *Job Scheduling Strategies for Parallel Processing (JSSPP 2013)*, Lecture Notes in Computer Science, vol. 8429, pp. 96–115, May 2013.

[50] Y. Kodama, S. Itoh, T. Shimizu, S. Sekiguchi, H. Nakamura, and N. Mori, "Imbalance of CPU Temperatures in a Blade System and its Impact for Power Consumption of Fans," *Cluster Computing*, vol. 16, no. 1, pp 27–37, Sep. 2013.

[51] G. V. Laszewski, L. Wang, A. J. Younge, and X. He. "Power-Aware Scheduling of Virtual Machines in DVFS-Enabled Clusters," *IEEE International Conference on Cluster Computing and Workshops (CLUSTER 2009)*, 10 pp., Aug.-Sep. 2009.

[52] Y. Ding, X. Qin, L. Liu, and T. Wang, "Energy Efficient Scheduling of Virtual Machines in Cloud with Deadline Constraint," *Future Generation Computer Systems*, vol. 50, pp. 62–74, Sep. 2015.

[53] I. Goiri, R. Beauchea, K. Le, T. D. Nguyen, M. E. Haque, J. Guitart, and R. Bianchini, "GreenSlot: Scheduling Energy Consumption in Green Datacenters," *ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, 20 pp., Nov. 2011.

[54] D. Machovec, B. Khemka, S. Pasricha, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, M. Wright, M. Hilton, R. Rambharos, and N. Imam, "Dynamic Resource Management for Parallel Tasks in an Oversubscribed Energy-Constrained Heterogeneous Environment," *25th Heterogeneity in Computing Workshop (HCW 2016)*, in 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 67–78, May 2016.

[55] D. Plummer, "An Ethernet Address Resolution Protocol," RFC 826, MIT-LCS, Nov. 1982.

[56] Address Resolution Protocol (ARP), ([Online]: http://linux-ip.net/html/ether-arp.html), accessed Mar. 2017.

[57] C. Tunc, D. Machovec, N. Kumbhare, A. Akoglu, S. Hariri, B. Khemka, and H. J. Siegel, "Value of Service Based Resource Management for Large-Scale Computing Systems," *Cluster Computing*, vol. 20, no. 3, pp. 2013–2030, Sep. 2017.

Chapter 3

Utility-Based Resource Management in an Oversubscribed Energy-Constrained Heterogeneous Environment Executing Parallel Applications[2]

3.1. 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

---

[2] The material in this chapter appeared in [38]. This work was done jointly with former Ph.D. student Bhavesh Khemka and Christopher Blandin. The full list of co-authors for this work is at [38]. 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.

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., [2-5]). 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 3.2, we define the HPC environment and problem we are addressing. Section 3.3 explains the resource management techniques that are utilized. The setup for our simulated environment is detailed in Section 3.4. The simulation analyses and comparisons are presented in Section 3.5. An experiment that we performed on a testbed system has its setup and results shown in Section 3.6. In Section 3.7, we discuss related work. Finally, in Section 3.8 we conclude and discuss future work.

3.2. Environment and Problem Description

3.2.1. Compute System Model

We modeled an environment where the compute system is composed of heterogeneous clusters of nodes, as shown in Figure 18. 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].

3.2.2. 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).

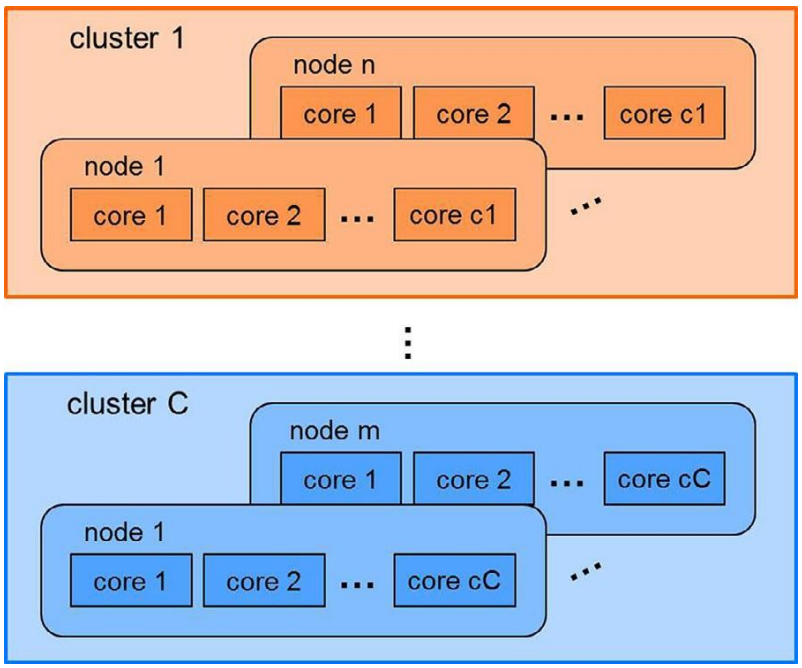


Figure 18. A compute system composed of C clusters. Cluster 1 has n nodes and cluster C has m nodes.

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., [13-15]). 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 Table 2. It is assumed that from past executions or experiments we have entries for certain levels of parallelism, i.e., for certain numbers of nodes. In 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 19, 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.

3.2.3. 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 20. 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.

3.2.4. 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.*

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

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

3.3. Resource Management

3.3.1. 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 3.3.3). Next, energy filtering (detailed in Subsection 3.3.8) attempts to improve energy efficiency by limiting the allocation options for tasks. Finally, one of the heuristics defined in Subsections 3.3.4, 3.3.5, or 3.3.6 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.

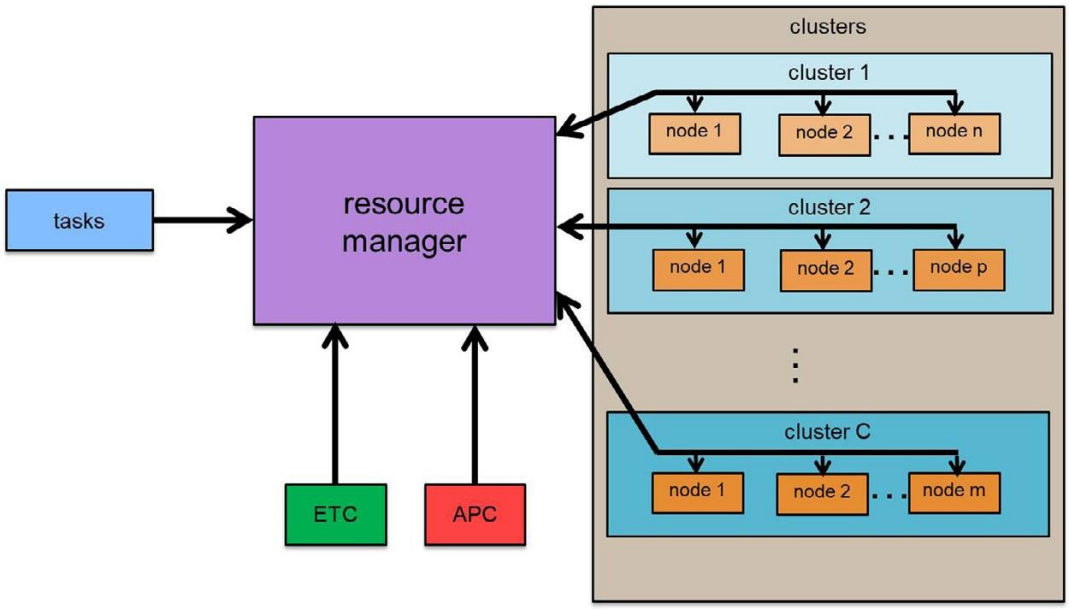


Figure 19. Flow for the proposed resource manager. Tasks enter the resource manager and are mapped to the nodes of clusters. Each task is mapped to the nodes of a single clusters.

3.3.2. 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:

$$resources\ allocated = execution\ time \times corse\ allocated. \quad (1)$$

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.

3.3.3. 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.

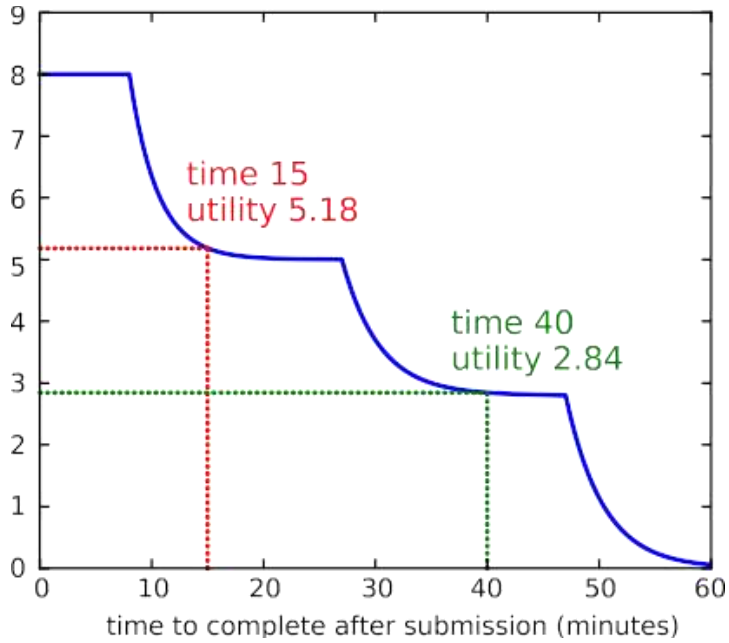


Figure 20. An example of a utility function for task 1. If task 1 completes at time 15, it earns 5.18 utility. If task 1 complete at time 40, it earns 2.84 utility.

3.3.4. Comparison Heuristics

3.3.4.1. 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 3.3.7.

3.3.4.2. 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.

3.3.4.3. 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.

3.3.4.4. 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.

3.3.4.5. 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.

3.3.5. Utility-Aware Heuristics

3.3.5.1. 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.

3.3.5.2. 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):

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

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

$$UPR = Util/resources\ allocated\ to\ the\ task \quad (4)$$

$$UPE = Util/energy\ consumed\ by\ the\ task \quad (5)$$

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.

3.3.5.3. 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 4, lines 2–4.

3.3.5.4. 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

Algorithm 4. Pseudo-Code for Max Util

```
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
```

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 4 for the Max UPE heuristic in lines 5–11. This algorithm also can use place-holders instead of permanent reservations, by replacing "permanent reservations" with "place-holders" in line 11 of Algorithm 4.

3.3.6. Metaheuristics

3.3.6.1. 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.

3.3.6.2. 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.

3.3.6.3. 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.

3.3.7. 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 4). 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 3.3.4, the utility-aware heuristics detailed in Subsection 3.3.5, and the metaheuristics described in Subsection 3.3.6).

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 21a, 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 21 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.

3.3.8. Energy Filtering

3.3.8.1. 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.

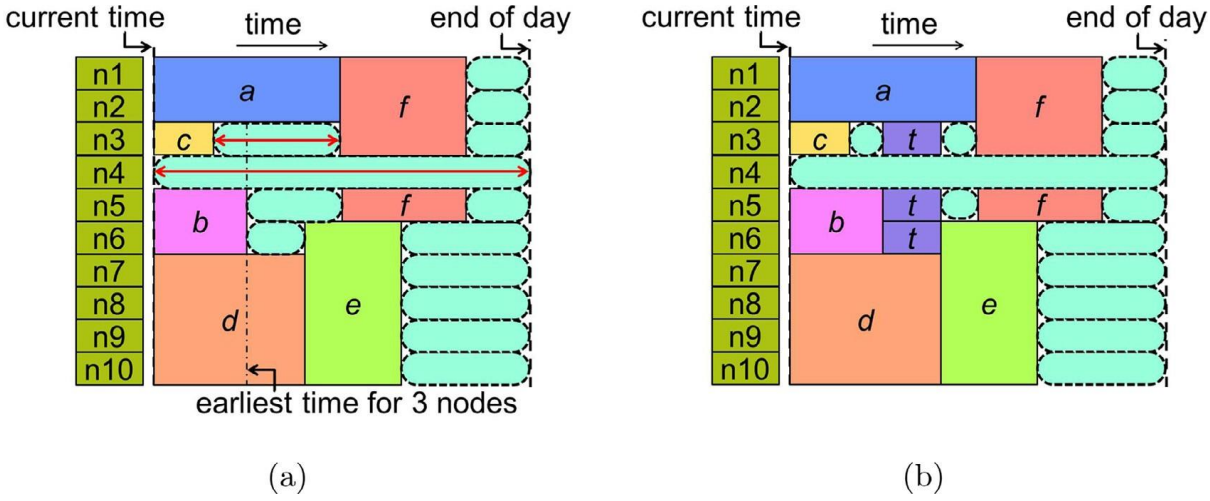
3.3.8.2. Energy-per-Task Filtering

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:

$$resources\ remaining = unallocated\ time\ remaining\ on\ node\ i \times cores\ on\ node\ i, \quad (6)$$

$$estimated\ number\ of\ tasks\ remaining = min\left(\frac{resource\ remaining}{average\ resources\ used}, \frac{energy\ remaining}{estimated\ number\ of\ tasks\ remaining}\right), \quad (7)$$

$$energy\text{-}per\text{-}task\ budget = leniency\ factor \times \frac{energy\ remaining}{estimated\ number\ of\ tasks\ remaining}. \quad (8)$$



(a) (b)

Figure 21. 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 3.3.7.

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.

3.3.8.3. 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:

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

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.

3.4. Simulation Setup

3.4.1. 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 3.4.2. 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.

3.4.2. Generation of Compute System and a Synthetic Workload

3.4.2.1. 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 3 (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 3.2.3) 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

Table 3. 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%

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 22. 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 4 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

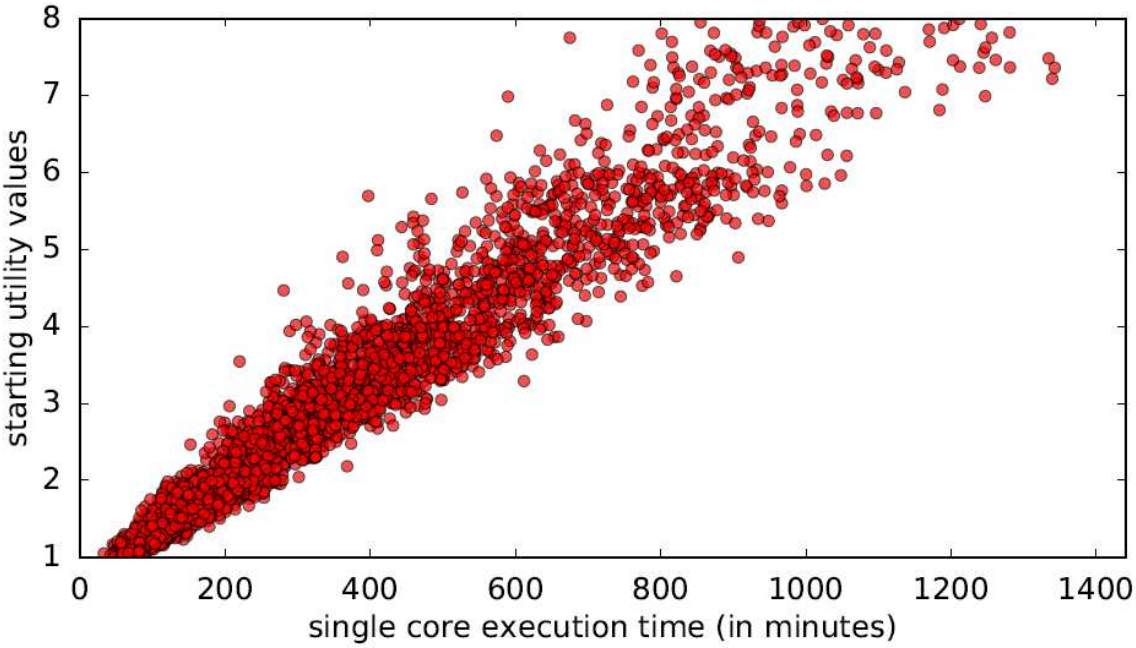


Figure 22. The correlation between single core execution time and starting utility for the 48 simulation scenarios, each with 100 task types, for a total of 4,800 points.

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.

Table 4. Core distribution of tasks.

percentage of tasks	min	max
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

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:

$$T(n) = \begin{cases} \sigma + \frac{A+A\sigma-\sigma}{n}, & 1 \le n \le A + A\sigma - \sigma \\ \sigma + 1, & n > A + A\sigma - \sigma \end{cases} \quad (10)$$

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.

3.4.3. 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 [24, 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 3.4.2. 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 3.4.2.

3.4.4. Resource Management Parameters

3.4.4.1. 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.

3.4.4.2. 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 3.2.3). 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 Figure 23 and Figure 24 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.

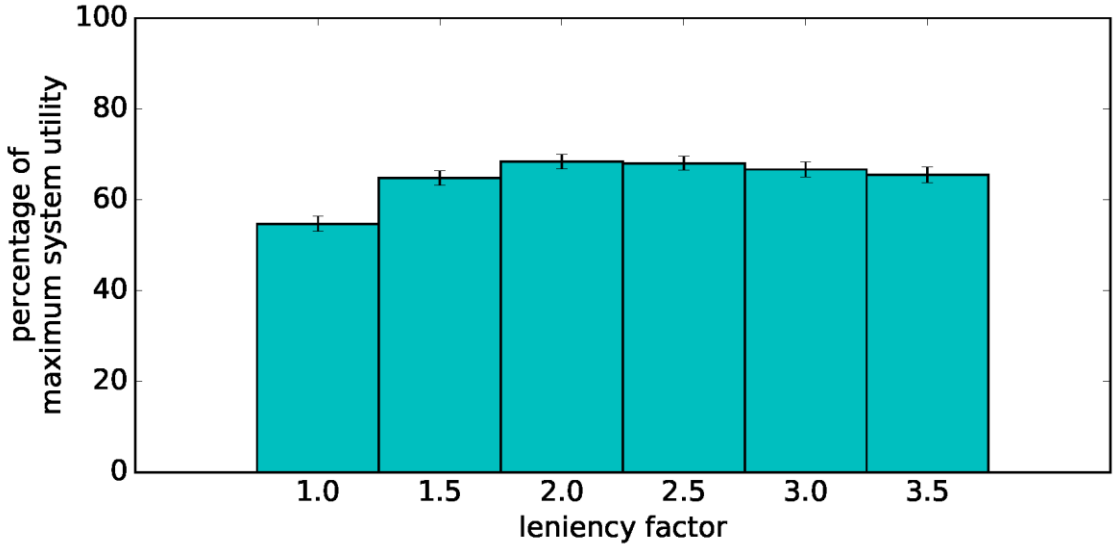


Figure 23. A range of energy leniency factors using energy-per-task filtering for the Max UPR with place-holders heuristic.

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.

3.5. Simulation Results

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

In Figure 25a, 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 25b with an energy constraint of 12 gigajoules. Results using the energy filters are not shown for the UPE heuristic because they had identical

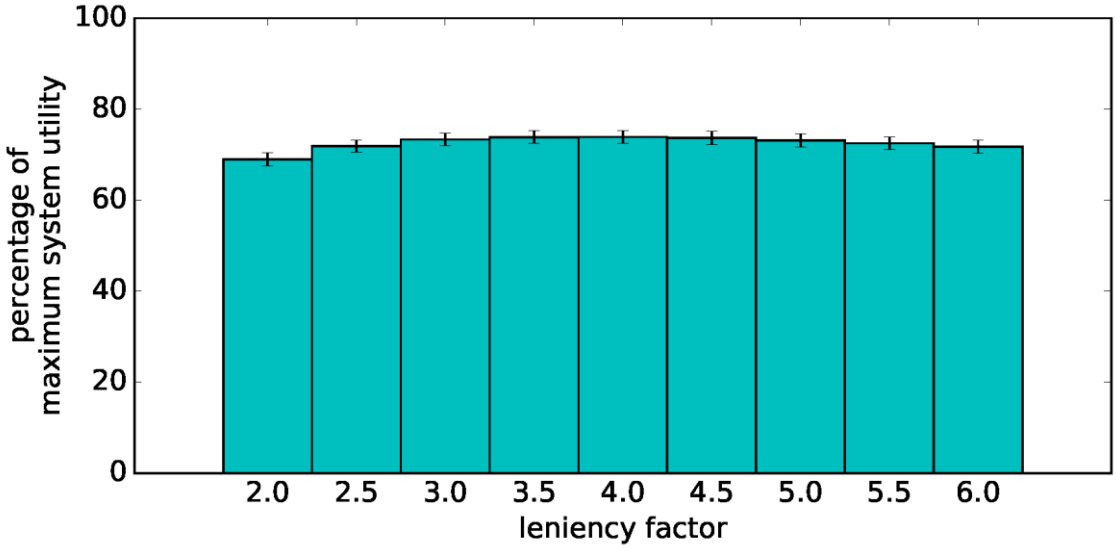


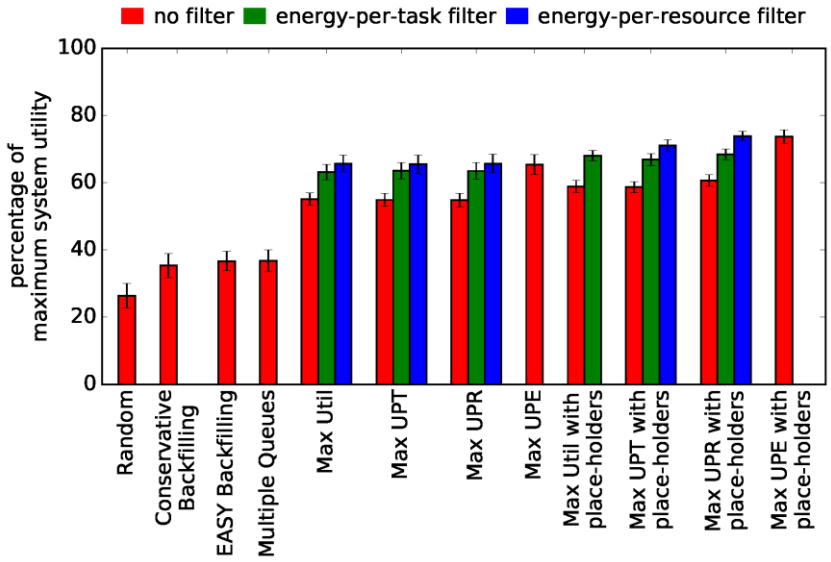
Figure 24. A range of energy leniency factors using energy-per-resource filtering for the Max UPR with place-holders heuristic.

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 25 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.

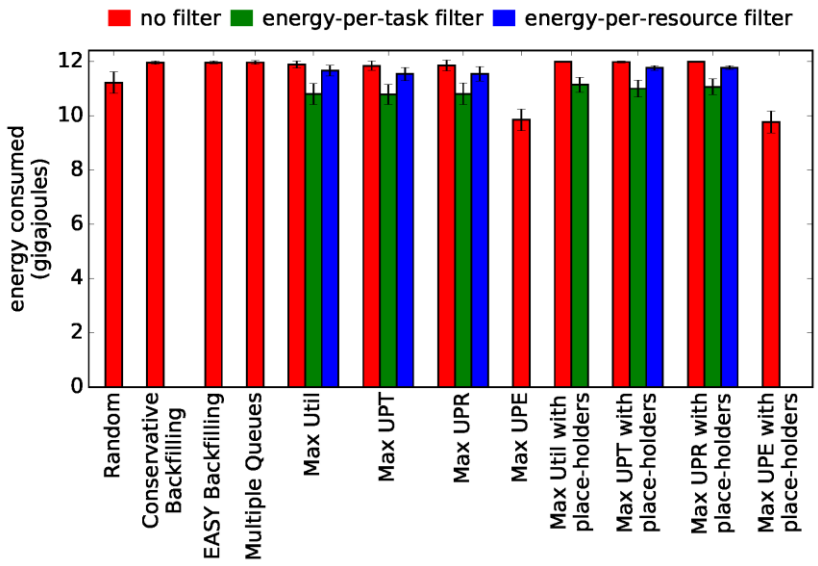
The utility-aware heuristics (Max Util, Max UPT, Max UPR, and Max UPE) that we proposed to solve this problem are able to earn significantly more utility than the comparison heuristics from the literature that do not consider utility and make permanent reservations instead of using place-holders (EASY 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 Figure 25a and Figure 25b 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 22). 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 26. The relative performance of Max UPR with place-holders and Max UPE with place-



(a)



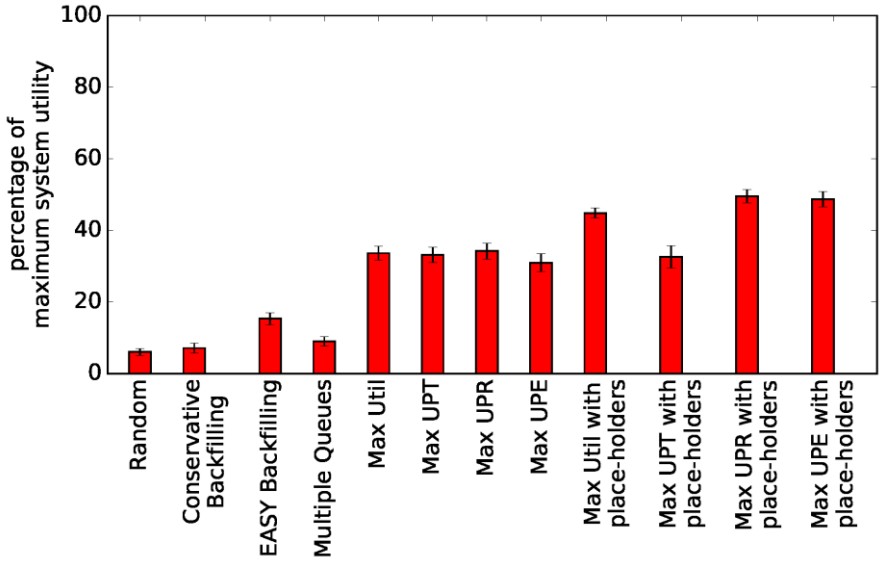
(b)

Figure 25. 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.

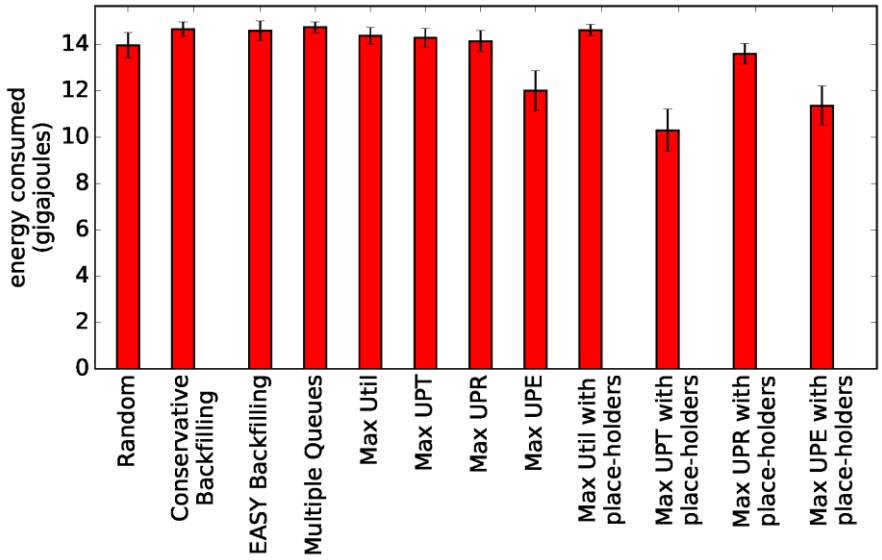
holders is comparable to the results for a mean of 5,000 task arrivals per day, but the performance of Max UPT with place-holders has degraded such that the other place-holder heuristics perform better with no overlapping 95% confidence intervals, obtaining up to 50% more utility in the case of Max UPR 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 26a is compared with the results in Figure 25a, 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 Figure 25a and Figure 26a, the Max UPE with place-holders heuristic consumes less energy as seen in Figure 25b and Figure 26b, where the average energy consumed by Max UPR with placeholders increases by 20% when compared to Max UPE with place-holders.



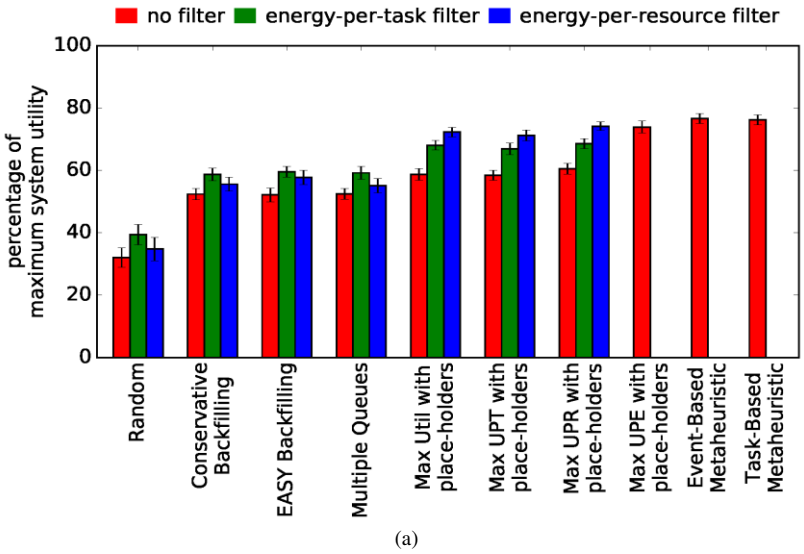
(a)



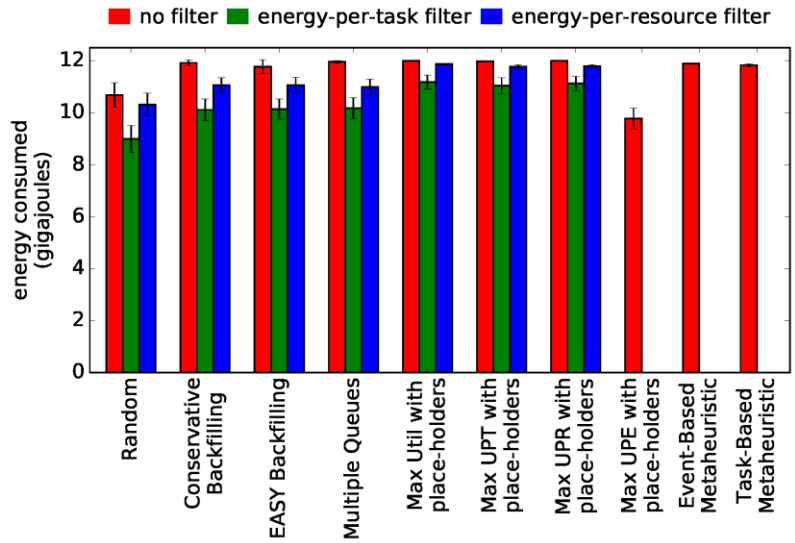
(b)

Figure 26. 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.

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.



(a)



(b)

Figure 27. Results for a mean of 5,000 tasks arriving per day. A comparison of the two metaheuristics with the heuristics from Subsection 3.3.6, 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.

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 25. 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 26, 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 25 and the Max UPR with place-holders heuristic with energy-per-resource filtering took 80 seconds in the larger simulations shown in Figure 26. 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.

3.5.2. 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 27. 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 3.5.4. 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 27 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.

3.5.3. 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 28. 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.

3.5.4. Energy Constraint Analysis

In Figure 29a, 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 29b. The confidence intervals in Figure 29 are based on the 48 simulation trials.

It can be seen in Figure 29a 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

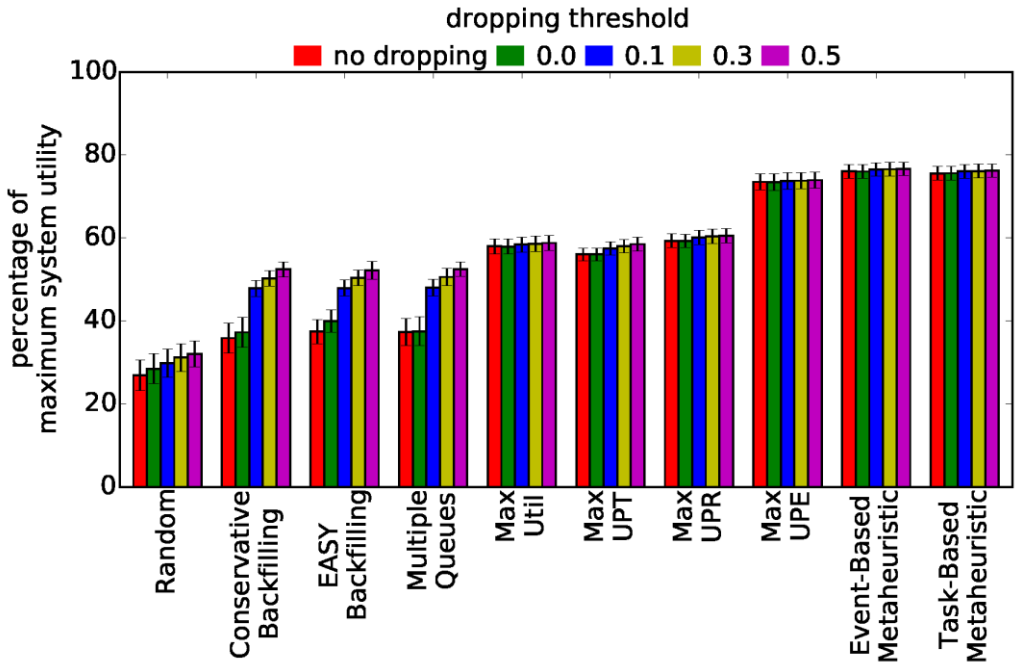


Figure 28. Results for a mean of 5,000 tasks arriving per day, where the dropping threshold is varied, the energy constraint is 12 gigajoules, and none of the heuristics utilize the energy filtering techniques. The percentage of maximum utility earned for each environment with 95% mean confidence intervals is shown.

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

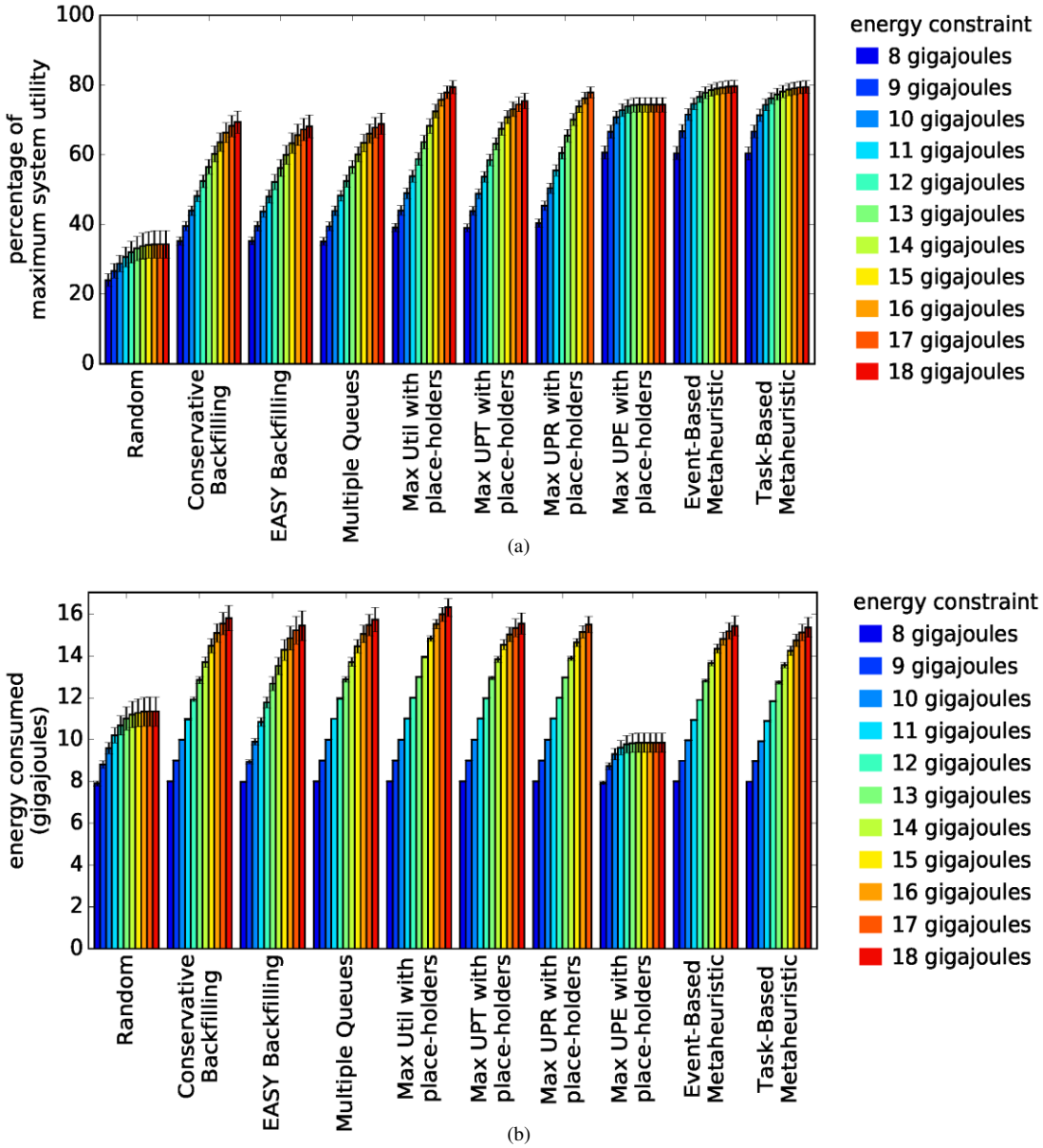
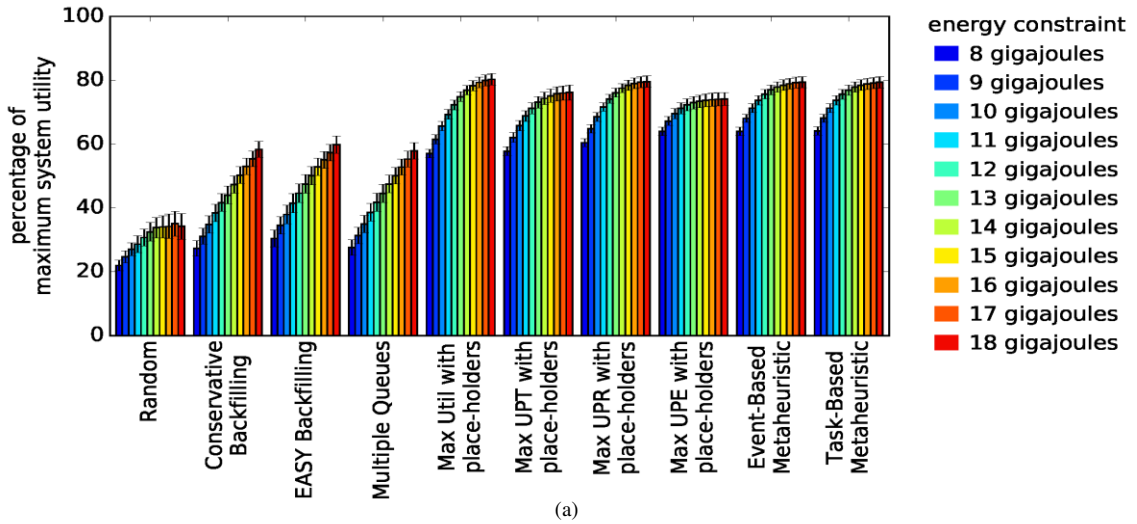
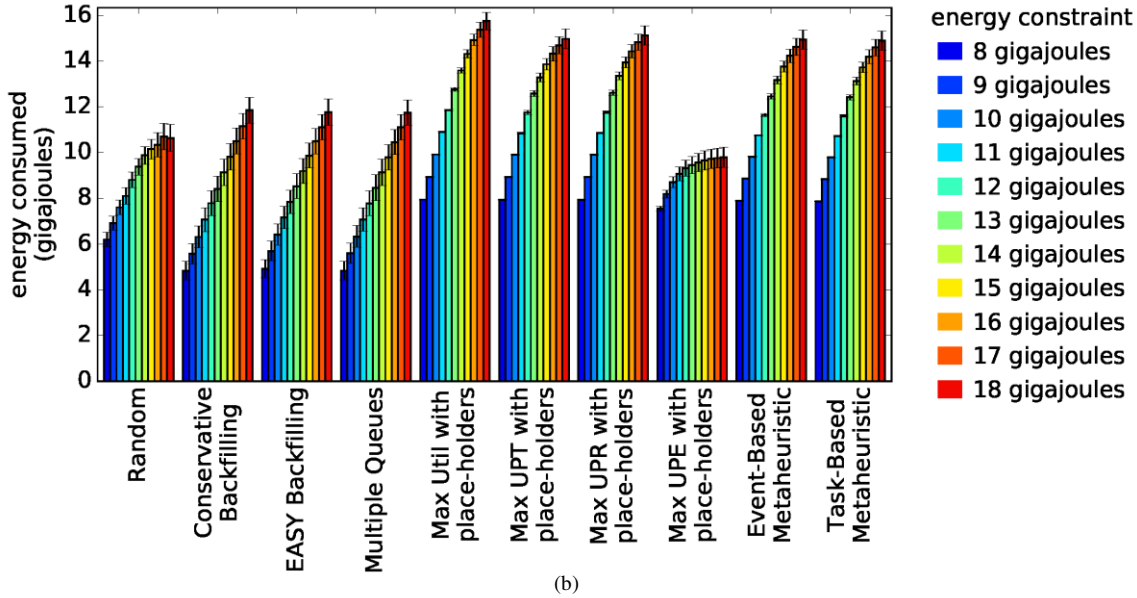


Figure 29. Results for a mean of 5,000 tasks arriving per day where the energy constraint is varied from 8 gigajoules to 18 gigajoules. None of the heuristics utilize the energy filtering techniques. (a) Percentage of maximum utility for a variety of energy constraints with no energy filter with 95% mean confidence intervals. (b) Energy consumption for a variety of energy constraints with no energy filter with 95% mean confidence intervals.



(a)



(b)

Figure 30. 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.

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.

3.5.5. Impact of Integrating an Energy Filter

Results for the same set of environments using the energy-per-resource filter are shown in Figure 30. Figure 30a 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 3.4 for the case with a 12 gigajoule energy constraint. Figure 30b 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.

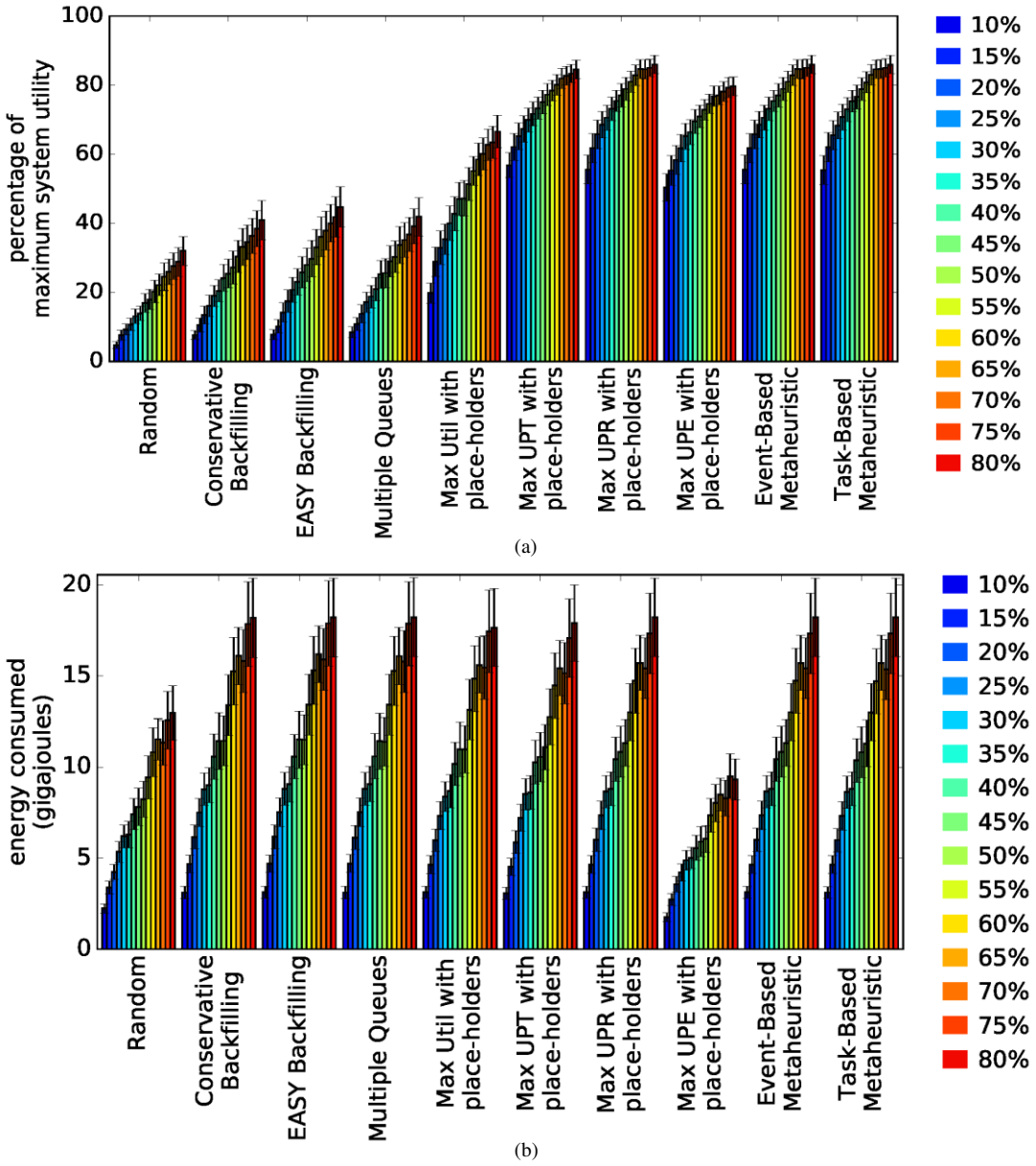


Figure 31. Results when tasks are based on a trace from the Curie supercomputer and there is no energy constraint. In these environments, the size of the system is varied from 10% to 80% of the original Curie system. (a) Percentage of maximum utility earned for each environment with 95% mean confidence intervals. (b) Energy consumption for each environment with 95% mean confidence intervals.

3.5.6. Analyses with Curie Workload Arrival Trace

3.5.6.1. Results without an Energy Constraint

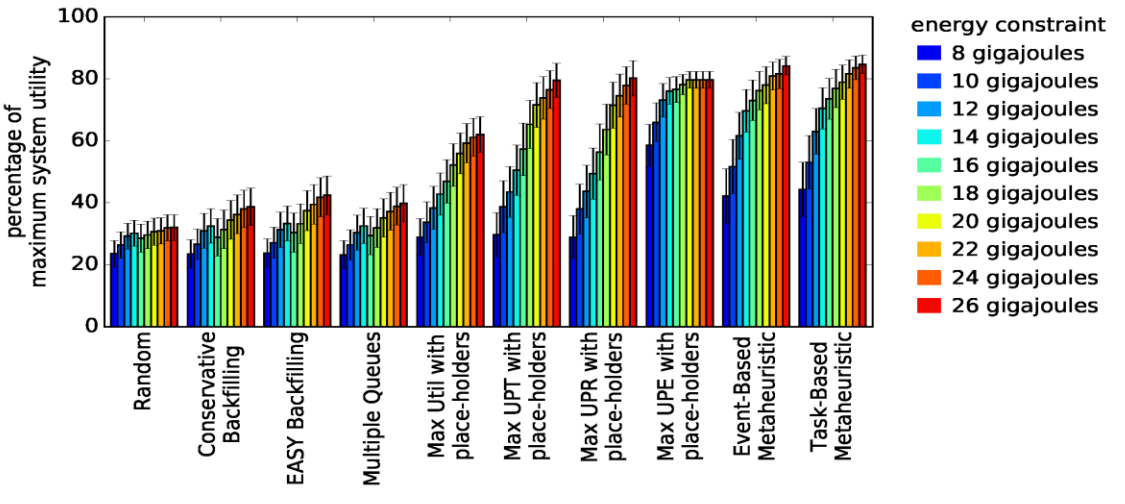
Figure 31a shows the percentage of maximum system utility earned in 15 environments using the workload generated from the Curie system trace. The different cases shown for each heuristic

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 31b. The confidence intervals in Figure 31 are based on the 48 simulation trials.

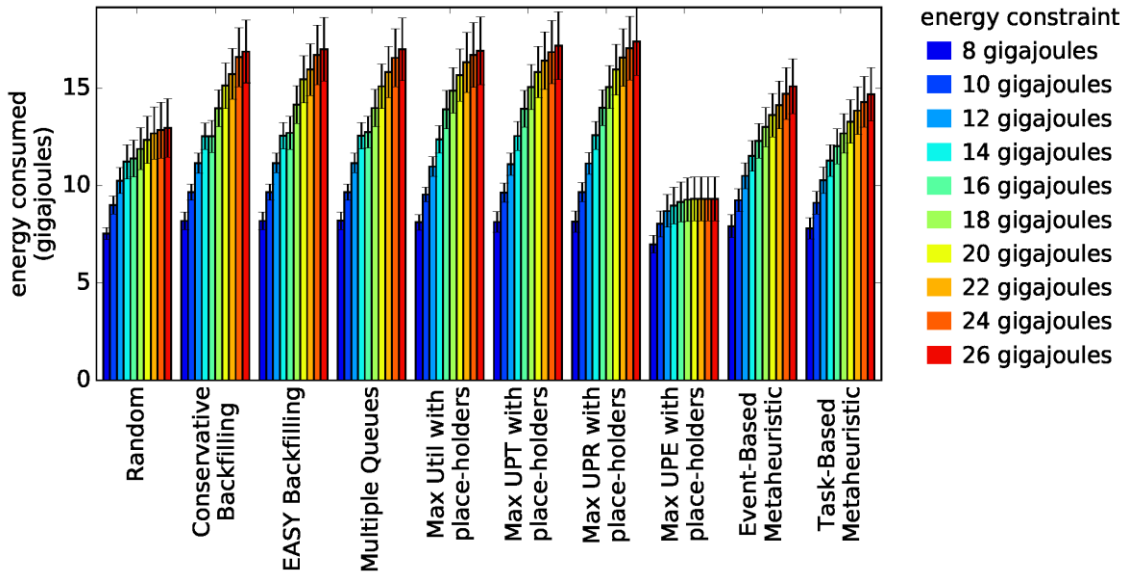
Figure 31a shows that the utility earned by the metaheuristics, Max UPT, and Max UPR is the highest among the heuristics. This makes sense because this environment is not energy constrained and the heuristics that do not consider the energy consumed by a task perform the best. The reason for the poor 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.

3.5.6.2. Results with an Energy Constraint

Figure 32 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 32a, is Max UPE. This is because the



(a)



(b)

Figure 32. 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.

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 32b) is lowest for the Max UPE heuristic. The other heuristics have energy consumption that stays closer to the energy constraint.

3.5.7. 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 Figure 29 and Figure 30. We also evaluated and compared several of our heuristics on a testbed system as discussed in the next section.

3.6. Experiment

3.6.1. 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 5 (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.

Table 5. 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

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 6 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.

Table 6. 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	1675	750	552	430
MG.8	8	1100	712	n.a.	n.a.
MG.16	16	702	500	380	326
IS_CG.8	8	1039	900	700	1366

Table 7. 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

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 5. 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 7 represents the estimated energy consumption to compute (EEC) for all the benchmarks on different clusters. In our experimental study, each benchmark represents a task.

3.6.2. 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

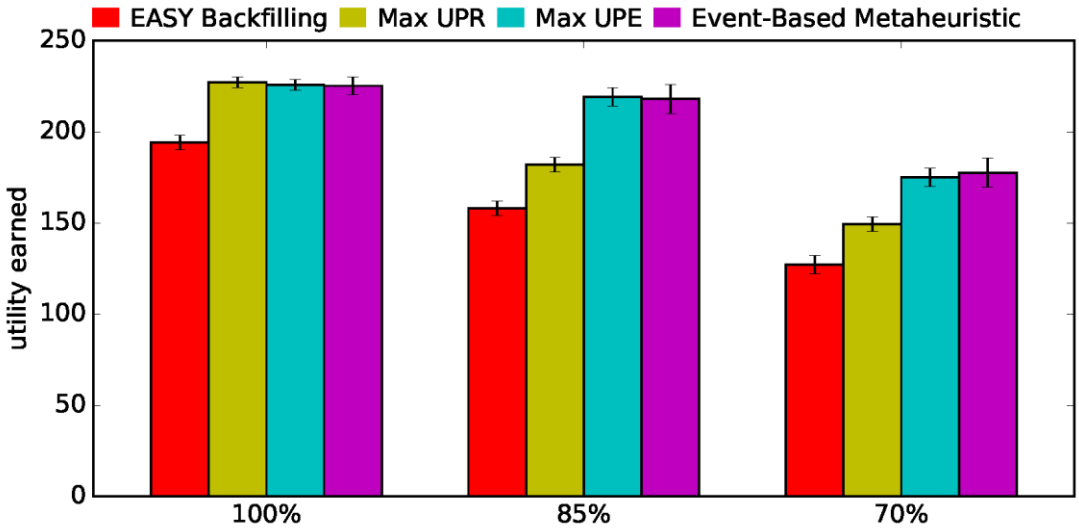


Figure 33. Utility earned versus energy constraint, as percent of maximum allowed, where maximum is 2.8 megajoules. The standard deviation is shown for each bar.

heuristics to identify the next set of tasks for scheduling. We considered dynamic energy as the energy metric for all the heuristics.

3.6.3. 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 33 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.

3.7. 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, 28−30]). 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 [33−36]. 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.

3.8. 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.

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, and S. Poole, "Utility Functions and Resource Management in an Oversubscribed Heterogeneous Computing Environment," IEEE Transactions on Computers, vol. 64, no. 8, pp. 2394–2407, 2015.

[2] P. Bohrer, E.N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, and R. Rajamony, "The Case for Power Management in Web Servers," *Power Aware Computing*, Series in Computer Science, pp. 261–289, 2002.

[3] I. Rodero, J. Jaramillo, A. Quiroz, M. Parashar, F. Guim, and S. Poole, "Energy-Efficient Application-Aware Online Provisioning for Virtualized Clouds and Data Centers," *International Green Computing Conference*, pp. 31–45, 2010.

[4] I. Ahmad, R. Arora, D. White, V. Metsis, and R. Ingram, *Energy-Constrained Scheduling of DAGs on Multi-Core Processors*, Springer Berlin Heidelberg, Berlin, Heidelberg, 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, and 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, and S. Poole, "Utility Maximizing Dynamic Resource Management in an Oversubscribed Energy-Constrained Heterogeneous Computing System," *Sustainable Computing: Informatics and Systems*, vol. 5, pp. 14–30, 2015.

[7] M. R. Garey and 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*, Lecture Notes in Computer Science, vol. 949, 1995, pp. 295–303.

[9] A. W. Mu'alem and D. G. Feitelson, "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling," *IEEE Transactions on Parallel Distributed Systems*, vol. 12, no. 6, pp. 529–543, 2001.

[10] B. Khemka, D. Machovec, C. Blandin, H. J. Siegel, S. Hariri, A. Louri, C. Tunc, F. Fargo, and A. A. Maciejewski, "Resource Management in Heterogeneous Parallel Computing Environments with Soft and Hard Deadlines," *11th Metaheuristics International Conference (MIC 2015)*, 10 pp, 2015.

[11] D. Machovec, B. Khemka, S. Pasricha, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, M. Wright, M. Hilton, R. Rambharos, and N. Imam, "Dynamic Resource Management for Parallel Tasks in an Oversubscribed Energy-Constrained Heterogeneous Environment," *25th Heterogeneity in Computing Workshop (HCW 2016)*, 2016 International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2016), pp. 67–78, 2016.

[12] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation, "Device power state definitions,"

September in Advanced Configuration and Power Interface Specification, Rev. 3.0, 2004, accessed: Aug. 2015.

[13] H. Barada, S. M. Sait, and N. Baig, "Task Matching and Scheduling in Heterogeneous Systems Using Simulated Evolution," *10th IEEE Heterogeneous Computing Workshop (HCW 2001)*, pp. 875–882, 2001.

[14] A. Ghafoor and J. Yang, "A Distributed Heterogeneous Supercomputing Management System," *IEEE Computer*, vol. 26, no. 6, pp. 78–86, 1993.

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

[16] Colorado State University ISTeC Cray High Performance Computing System, ([Online]: http://istec.colostate.edu/activities/cray), accessed: Nov. 2016.

[17] O. H. Ibarra and C. E. Kim, "Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors," *Journal of the ACM*, vol. 24, no. 2, pp. 280–289, 1977.

[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, and B. Yao, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, 2001.

[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*, vol. 59, no. 2, pp. 107–131, 1999.

[20] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems," *Journal of Parallel and Distributed Computing*, Special Issue on Software Support for Distributed Computing, vol. 59, no. 2, pp. 107–131, 1999.

[21] S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, and S. Ali, "Representing Task and Machine Heterogeneities for Heterogeneous Computing Systems," *Tamkang Journal of Science and Engineering*, Special Tamkang University 50th Anniversary Issue, vol. 3, no. 3, pp. 195–207, 2000.

[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: Nov. 2015.

[24] D. G. Feitelson, D. Tsafrir, and D. Krakov, "Experience with Using the Parallel Workloads Archive," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2967–2982, 2014.

[25] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux Virtual Machine Monitor," *The Linux Symposium*, vol. 15, pp. 225–230, 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: Aug. 2015.

[27] NAS Parallel Benchmarks, NAS-NPB, ([Online]: https://www.nas.nasa.gov/publications/npb.html), accessed: Aug. 2015.

[28] D. Klusaccek and H. Rudova, "Performance and Fairness for Users in Parallel Job Scheduling," *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, vol. 7698, pp. 235–252, 2012.

[29] A. Mishra, S. Mishra, and D.S. Kushwaha, "An Improved Backfilling Algorithm: SJF-B," *International Journal on Recent Trends in Engingeering & Technology*, vol. 5, no. 1, pp. 78–81, 2011.

[30] Y. Yuan, Y. Wu, W. Zheng, and K. Li, "Guarantee Strict Fairness and Utilize Prediction Better in Parallel Job Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 4, pp. 971–981, 2014.

[31] B. Ravindran, E. D. Jensen, and P. Li, "On Recent Advances in Time/Utility Function Real-Time Scheduling and Resource-Management," *8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pp. 55–60, 2005.

[32] M. Snir and D. A. Bader, "A Framework for Measuring Supercomputer Productivity," *International Journal of High Performance Computing Applications*, vol. 18, no. 4, pp. 417–432, 2004.

[33] E. Jensen, C. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Systems," *IEEE Real-Time Systems Symposium*, pp. 112–122, 1985.

[34] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Real-Time Operating Systems," *IEEE Transactions on Software Eng*ineering, vol. 30, no. 9, pp. 613–629, 2004.

[35] C. B. Lee and A. E. Snavely, "Precise and Realistic Utility Functions for User-Centric Performance Analysis of Schedulers," *International Symposium on High Performance Distributed Computing (HPDC 07)*, pp. 107–116, 2007.

[36] D. Vengerov, L. Mastroleon, D. Murphy, and N. Bambos, "Adaptive Data-Aware Utility-Based Scheduling in Resource-Constrained Systems," Technical Report 2007-164, Sun Microsystems, Inc., 2007.

[37] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan, "Scheduling of Parallel Jobs in a Heterogeneous Multi-Site Environment," *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, vol. 2862, pp. 87–104, 2003.

[38] D. Machovec, B. Khemka, N. Kumbhare, S. Pasricha, A. A. Maciejewski, H. J. Siegel, A. Akoglu, G. A. Koenig, S. Hariri, C. Tunc, M. Wright, M. Hilton, R. Rambharos, C. Blandin, F. Fargo, A. Louri, and N. Imam, "Utility-Based Resource Management in an Oversubscribed Energy-Constrained Heterogeneous Environment Executing Parallel Applications," *Parallel Computing*, vol. 83, pp. 48-72, Apr. 2019.

Chapter 4

Preemptive Resource Management for Dynamically Arriving Tasks in an Oversubscribed Heterogeneous Computing System[3]

4.1. Introduction

We study resource management for high performance computing (HPC) environments. The workload of such a system can be dynamic, i.e., the arrival pattern of tasks in the system is not known in advance. In addition, HPC systems are often oversubscribed, i.e., there are not enough resources in the system to begin executing each task as soon as it arrives. These systems also often have many different kinds of resources resulting in heterogeneity in the system, where different tasks have different execution characteristics on different resources.

To create strategies for effective resource management, it is necessary to define performance measures for the system. In some systems, these measures include utilization, fairness, or

---

[3] The material in this chapter appeared in [31]. The full list of co-authors for this work is at [31]. 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 nonexclusive, 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-publicaccessplan). This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory (ORNL), supported by the Extreme Scale Systems Center at ORNL, which is supported by the Department of Defense (DoD); and by NSF Grant CCF-1302693. This work also utilized CSU's ISTeC Cray system, which is supported by the National Science Foundation (NSF) under grant number CNS-0923386.

makespan (e.g., the systems studied in [17, 19, 25, 26]). In an oversubscribed HPC system, utilization is not an effective metric because the system should always be close to 100% utilization. In addition, many environments have tasks that have differing relative importance, making fairness an ineffective performance measure. Finally, makespan is not a good performance measure in oversubscribed environments where some tasks may not be executed. For example, if no tasks are executed then the makespan will be zero.

To quantify the performance of an oversubscribed heterogeneous system, we make use of utility functions that are monotonically decreasing in value based on task completion time [15]. We consider tasks with the highest possible starting utility to be critical tasks and all other tasks to be non-critical tasks. For example, users could pay to have higher utility for their tasks or in military environments, certain tasks are considered more "mission critical." The performance of the system is then measured by considering the total utility earned by each task over a period of time. Because the utility earned over this period can be significantly affected by the percentage of critical tasks that are executed, the design of heuristics that are able to more effectively prioritize execution of these tasks is essential.

Because the system is dynamic and oversubscribed, critical tasks may have their execution delayed or prevented by less important non-critical tasks that were already in the workload before the critical tasks arrived. These non-critical tasks would not have been scheduled if the resource manager had been aware that critical tasks would be arriving to the system before completion of the non-critical tasks. In addition, choosing to not execute the non-critical tasks is not an ideal solution because they may not result in the delay of critical tasks (i.e., they may finish executing before any critical tasks arrive in the system). An effective way to address this issue is to allow the preemption of tasks that are already executing in the system. In an environment with preemption,

critical tasks can start executing quickly even if the system is already completely allocated to non-critical tasks.

In this work, we study an environment where the workload is comprised of serial tasks with short execution times (average of 50 minutes for most tasks). Such environments exist in some DoD (Department of Defense) and DOE (Department Of Energy) systems. Preemption is commonly used in these and similar environments, such as those used for MapReduce where an application is split into many similar small serial tasks [6]. In addition, the preemption overhead of serial tasks with short execution times is insignificant compared to the overhead that would occur when preempting resource intensive parallel tasks with much larger memory requirements [3].

Another environment where a large number of serial jobs could occur is an enterprise datacenter. In HPC environments designed for parallel tasks, it is often the case that multiple tasks are not assigned to the same node (even in cases where many cores on the node will be idle). This assumption was used in [21], where resource management heuristics were designed for a similar environment with parallel tasks and no preemption. Because utilizing fewer than the total number of cores in a node is inefficient, it is logical to have a separate system where scheduling is done at a core-level to execute only serial tasks when a large number of serial tasks is expected in the workload.

The specific systems that we consider in this study are designed to model production environments that exist in military, government, and industrial situations. These systems were constructed based on discussions with researchers from Oak Ridge National Laboratory (ORNL) and the United States DoD. ORNL and DoD are examining an implementation of the resource

management heuristics proposed in this paper and have environments with serial tasks like the ones simulated in this study.

There are many environments that can be considered when studying resource allocation for HPC systems. In any given HPC environment, (a) tasks may be parallel or serial and may have dependencies on one another; (b) tasks may have varying importance; (c) preempting and resuming tasks may require significant overhead; (d) there may be homogeneity or heterogeneity among clusters of the system and each cluster may contain homogeneous or heterogeneous nodes; (e) the workload may cause the system to be oversubscribed; (f) tasks may arrive dynamically or may be given to the scheduler in a single batch; and (g) the execution time of tasks may not be known in advance. The parameters described above only cover a subset of the total problem space of resource management HPC environments. To make this study tractable, we focused on a specific environment of ORNL and DoD interest. The techniques proposed in this study were designed specifically for this environment and may not apply to other environments.

A specific set of important assumptions made in this paper includes that: (a) tasks are independent and serial; (b) some tasks are significantly more important than others; (c) the overhead of preempting and resuming a task is insignificant compared to the execution time of tasks; (d) the compute system is made up of heterogeneous clusters where each cluster is comprised of homogeneous nodes; (e) the system is oversubscribed; (f) tasks arrive dynamically throughout the day and the scheduler does not have future knowledge of the set of tasks that will arrive; and (g) an estimate of the execution time of each task type on a node of each of the clusters is known to the scheduler.

We designed and implemented three preemption techniques that can be applied to previously studied utility-aware heuristics. These heuristics are Max Utility and Max Utility-per-Time, which

have been studied in a serial environment without preemption [15]. These heuristics are evaluated with preemption and without preemption in this study. They are also compared with the Random and FCFS heuristics. The primary contributions of this work are:

- The design and implementation of multiple techniques for applying preemption to utility-aware heuristics that have been effective in similar environments without preemption.
- An extensive analysis of the effectiveness of the proposed preemption techniques in many different simulated environments.

The rest of this paper is organized as follows. In Section 4.2, we define the studied environment and problem in detail. The proposed heuristics with preemption and other resource management techniques used in this study are detailed in Section 4.3. Section 4.4 describes the procedure that was used to generate the specific systems and workloads that were simulated and Section 4.5 presents the results of those simulations. Related work is discussed in Section 4.6. Finally, we conclude and discuss ideas for future work in Section 4.7.

4.2. Environment Description

4.2.1. System Model

The environment is one where the compute system is composed of heterogeneous clusters of compute resources. The architecture of cores varies across clusters, but each cluster has a fixed number of homogeneous cores. The execution characteristics of a task are identical on any cores of the same cluster. In this study, each serial task is assigned to a single core in one of the clusters.

4.2.1.1. System Workload Characteristics

The system workload is made of up of serial tasks that arrive dynamically. The environment is oversubscribed, meaning that it is not possible for every task to earn its maximum utility because

some tasks will be delayed due to the number of tasks competing for resources. We assume that the tasks are independent (i.e., there is no communication between different tasks). For this study, we assume that memory interference between two tasks executing on the same multi-core node is negligible. In this workload, tasks arrive in bursts. Each burst consists of a number of tasks with the same task type. This is meant to model an environment where users often submit many similar jobs to the system at once, such as found in some ORNL and DoD environments.

Each task has a task type that is known when it arrives. A task type is used to group tasks with similar characteristics together. Each task specifies its task type, its utility function, a flag that states whether it can be preempted, and a flag that states whether it can preempt other tasks. The execution time of each task type is defined using an Estimated Time to Compute (ETC) matrix [5, 23]. This matrix specifies the estimated execution time of each task type on a core of each cluster type. The ETC values can be based on user supplied information, experimental data, or task profiling and analytical benchmarking (e.g., [1, 9, 10, 16, 24, 30]). Determination of ETC values is a separate research problem; the assumption of such ETC information is a common practice in resource allocation research (e.g., [4, 10, 16, 18, 27, 29]). For environments such as those found in many ORNL and DoD systems, this matrix can be populated using historical data or experiments for each task type. This assumption is true, in general, for many production environments such as military, government, and industrial. These execution times are assumed to be deterministic for this study. An example of an ETC matrix that shows the execution time of task types in minutes for three clusters and four task types can be seen in Table 8. Task type 2 has its fastest execution on cluster A, task types 3 and 4 have their fastest execution times on cluster B, and task type 1 has its fastest execution time on cluster C. This type of heterogeneity where one of the clusters is not

the best for all task types is called inconsistent heterogeneity [5, 16]. In this study, the heterogeneity of our simulated systems is inconsistent.

4.2.2. Utility Functions

Utility functions are a flexible measure of the importance of tasks. Utility functions are monotonically decreasing functions of a task's completion time that are used to define the utility that a task earns upon completing its execution. This is because in this problem domain the later that information is delivered the less useful it is. An example of a possible utility function is shown for some task 1 in Figure 34. In this case, the utility function has a starting utility of 8, and decays to zero over approximately 60 minutes after the arrival of task 1. This utility function is generated using the model for utility functions presented in [15]. In many cases, it is suitable to simplify these potentially complex utility functions by using simple functions such as step functions or ones with linear decay [22].

4.2.3. Preemption

Each task that arrives in the system has two flags specifying its behavior involving preemption to the resource manager. One of the flags determines whether or not the task can be preempted by other tasks. The other flag determines whether a task can preempt other tasks. These flags act as a constraint on the resource manager. In addition, we do not limit the number of times a task can be preempted. Tasks are not preempted unless higher utility can be earned for another task. Any "starvation" of tasks is done intentionally to increase the utility earned by the system. This includes

Table 8. An Estimated Time to Compute (ETC) Matrix

task type	execution time (minutes)		
	cluster A	cluster B	cluster C
1	50	56	37
2	43	51	85
3	12	6	23
4	51	45	96

cases where the scheduler drops a task with high starting utility due to missing its deadline. Because the system is oversubscribed, there may be situations where it is not possible for the optimal schedule to complete all critical tasks before their deadlines.

When a task is preempted, we assume that its progress is saved and that it can be resumed at a later point on any core in the same cluster. The task cannot be resumed in another cluster because it can only be resumed on a core of the same architecture and where any task-specific data is available. We assume that the overhead for preempting and resuming a task is negligible. We make this assumption because all tasks in this system are serial and are assumed to have small memory requirements. The amount of memory allocated to a task is a significant factor in determining the amount of overhead that is needed to suspend and resume it [8]. This is because the majority of the overhead of preemption comes from the time needed to send a task's state to storage. We confirmed this with tests performed at ORNL. In some HPC systems, a task can only access the memory that is a part of the compute nodes that are assigned to the task. In these environments, such as the one we consider here, serial tasks may not have access to the amount of memory that would be required to generate a significant preemption overhead [3]. The techniques we designed can be built upon to add consideration of any non-negligible overhead. One way to add support

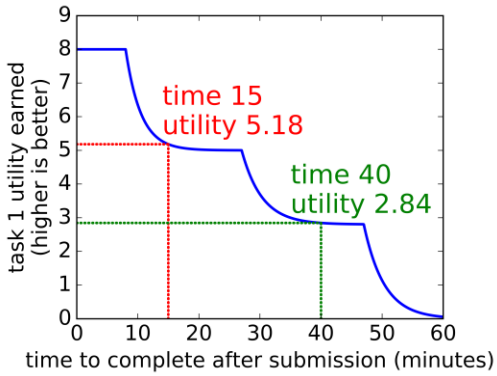


Figure 34. An example of a utility function for task 1. If task 1 were to finish its execution at time 15, it would earn 5.18 utility. If its execution completes at time 40, it will earn 2.84 utility. This figure was taken from our previous work [21].

for this would be for the preemption-capable heuristics to consider any expected overhead due to preempting a task as a part of the execution time of the preempting task. In addition, any overhead due to resuming a preempted task would be considered a part of the execution time of the resuming task.

4.2.4. Problem Statement

The resource manager has complete information about the tasks that are currently executing on the clusters as well as a list of the tasks that are waiting to be assigned. For each executing task, the resource manager can use the ETC matrix and the start time of each task to determine the estimated finish time of that task, which allows the heuristic to determine the estimated utility earned by that task. This information allows our resource manager to make intelligent allocation decisions using one of the heuristics described in Section 4.3.

We define the system utility earned during an interval of time as the sum of the utility earned by tasks that execute during that interval. In our simulations, a task will earn utility for an interval if any portion of the task executes during the interval. If only part of a task's execution occurs during this interval, then only a portion of the task's utility will be earned for that interval. For example, if a task were to complete 70% of its execution during an interval A and 30% of its execution during interval B, then 70% of the task's utility will be added to the system utility earned for interval A. The goal of our resource manager is to maximize the system utility earned by completing serial tasks over some interval of time.

4.3. Resource Management

4.3.1. Mapping Events

The set of mappable tasks contains all tasks that have arrived in the system excluding the tasks that have completed execution and that are currently executing. Any task in this set can be mapped to the available cores of the system. Mapping a task refers to the process of assigning, scheduling, or preempting the task to or from cores in the system. During a mapping event, the resource manager makes decisions for mappable tasks in the system. In this study, we consider the case where mapping events occur in one minute intervals, which is based on discussions with ORNL and DoD. At each mapping event, tasks are dropped (described in Subsection 4.3.2) and then one of the heuristics (described in Subsections 4.3.3, 4.3.4, and 4.3.5) is used to create a mapping of the tasks to the cores. Finally, this mapping is used to preempt and assign tasks.

4.3.2. Task Dropping

Our resource manager will not assign tasks to cores if they will earn zero utility. If at any point a task can never earn non-zero utility with any assignment, it is removed from the system. The resource manager will not assign a task to a core if that assignment will earn zero utility for that task.

4.3.3. Comparison Heuristics

4.3.3.1. Overview

For comparison, we consider two simple heuristics (Random and FCFS). These heuristics do not consider utility functions or the heterogeneity of the system.

4.3.3.2. Random

The Random heuristic begins with the set of mappable tasks and places them in a random ordering. It then iterates through the mappable tasks assigning each of them to a random idle core with the constraint that the task must earn non-zero utility on the randomly chosen core, or else a different random core is selected. If this is not possible, the task is left in the mappable set. After this assignment, the task is removed from the set of mappable tasks. This process is repeated until the set of mappable tasks is empty or no more assignments are possible.

4.3.3.3. FCFS

The FCFS (first come first served) heuristic places the mappable tasks in an order based on arrival time (i.e., the task with the earliest arrival time is considered first and the task with the most recent arrival time is considered last). It then takes each task in order and assigns it to an arbitrary idle core with the constraint that the task must earn non-zero utility. If this is not possible, the task is left in the mappable set. After this assignment, the task is removed from the set of mappable tasks. This process is repeated until the set of mappable tasks is empty or no more assignments are possible.

4.3.4. Utility-Aware Heuristics

4.3.4.1. Overview

All of our utility-aware heuristics are based on the two-phase concept of the Min-Min heuristic [11], a concept that has been used successfully in many heterogeneous environments (e.g., [5, 23]). The heuristics described in this section operate without preemption. The preemption techniques that we apply to them are described in Subsection 4.3.5.

Algorithm 5. Pseudo-Code for Max Util

```
1.  while mappable tasks is not empty and there is a valid
    assignment for a task in the mappable tasks:
2.      for each task t in mappable tasks:
3.          find idle core that maximizes Util for t;
4.          select task from mappable tasks with task/core
            combination that has the highest maximum Util;
5.          assign selected task to that core;
6.          remove task from mappable tasks;
7.  end while
```

4.3.4.2. Objective Functions

The task execution characteristics used by the objective functions in this study are the completion time (CT) and remaining execution time (RET) of a given assignment for a task. For a task that has not been previously executing in the system, RET is equal to the ETC entry for that task. If the task is currently executing or was previously preempted, its RET is equal to the ETC entry for the task minus the amount of time it has spent executing. Our heuristics attempt to greedily maximize either Utility (Util):

$$\text{Util} = \text{utility of the task at its CT}, \tag{1}$$

or Utility-per-Time (UPT):

$$\text{UPT} = \text{Util} / \text{RET}. \tag{2}$$

We use these objective functions to define two utility-aware heuristics called Max Utility (Max Util) and Max Utility-per-Time (Max UPT). These heuristics were applied successfully in our previous work without preemption [15, 21].

4.3.4.3. Maximum Util

Max Util, shown in Algorithm 5, operates in two phases. In the first phase (lines 2-4 of Algorithm 5), the heuristic considers each task individually and finds the idle core that maximizes its Util with the constraint that the task earns utility above the dropping threshold. This can be

achieved by considering a single idle core in each cluster because all of the cores in each cluster are homogeneous. The second phase of the heuristic (lines 5-6 of Algorithm 5) chooses the task/core pair from the first phase with the highest overall Util. The chosen task is then assigned to the chosen idle core to begin executing and is removed from the set of unmapped tasks. The heuristic then repeats, executing phase one and phase two until the unmapped tasks set is empty or there are no more possible assignments to make.

4.3.5. Preemption Techniques

4.3.5.1. Overview

As a part of this study, we designed and implemented three preemption techniques. All of these preemption techniques work by modifying how the heuristics described in Subsection 4.3.4

Algorithm 6. Pseudo-Code for Max Util Preempt Greedy

```
1.  while mappable tasks is not empty and there is a valid assignment for a task in the
    mappable tasks:
2.      for each task t in mappable tasks:
3.          if task t can preempt other tasks:
4.          then
5.              find cores that maximize Util for t that are idle or are executing a preemptible
                task where t has higher Util than the preemptible task;
6.              if an idle core was selected for t:
7.              then
8.                  choose the idle core;
9.              else
10.                 choose the core that is executing
                    a task with the minimum Util;
11.         else
12.             find idle cores that maximizes Util for t;
13.             select task from mappable tasks with task/core
                combination that has the highest maximum Util;
14.     if the core is not idle:
15.     then
16.         preempt the task it is executing and add it to the set of mappable tasks;
17.     assign selected task to that core;
18.     remove task from mappable tasks;
19. end while
```

interact with tasks that can preempt and tasks that are preemptible. We found no heuristics in the literature that used preemption in a heterogeneous environment.

4.3.5.2. Maximum Util Greedy Preemption

The Maximum Objective Function Greedy Preemption (Max Util Preempt Greedy) technique is similar to the Max Util technique (with no preemption) described above, except that it also considers all possible preemptions for each task in addition to idle cores. This technique is shown in Algorithm 6. Specifically, during the first phase of the heuristic (lines 2-12 of Algorithm 6) each task is considered individually. The core that maximizes its Util is found. If the core being considered is idle, then the same method used in the Max Util heuristic is used. If the core is executing a preemptible task, the task that is being considered can preempt other tasks, and the Util of the task being considered is greater than the Util of the currently executing task, then that pair represents a valid preemption. The Util of this pair is defined as the Util of the preempting task. The currently executing task's Util is only considered if during the first phase multiple pairs of task/core mappings are found that have the same Util for the preempting task. If multiple pairs are found during this first phase with the same Util, then the following strategy is used to select one. If any pair has an idle core, that pair is chosen. If there are multiple pairs with idle cores, one is chosen arbitrarily. If any pair does not use an idle core, choose the pair with the core that has the smallest Util for its preemptible task. After the best pair is found for each task, this technique has a phase two (lines 13-18 of Algorithm 6) where the pair with the highest overall Util is chosen and an assignment is made for the task/core pair (any necessary preemptions for this assignment will occur). This process repeats until there are no more unmapped tasks or there are no more valid pairs selected in the first phase.

4.3.5.3. Maximum Util Difference Preemption

The Util Difference is equal to the difference in Util of a preempting task and the task that is being preempted (when considering an idle core the Util and Util Difference are the same for the task being considered). The Maximum Util Difference preemption technique (Max Util Preempt Diff) attempts to greedily maximize the Util Difference for a task. It is identical to the Max Util Preempt Greedy technique except that all instances of Util are replaced with Util Difference throughout the execution of the heuristic. This heuristic is identical to the first technique in a homogeneous system, but may make different decisions in terms of which cluster to assign a task to in a heterogeneous system. For example consider a heterogeneous system with two cores: c1

Algorithm 7. Pseudo-Code for Max Util Preempt Pair

```
1.  while mappable tasks is not empty and there is a valid assignment for a task in the
    mappable tasks:
2.      for each task t in mappable tasks:
3.          if task t can preempt other tasks:
4.          then
5.              for each core c that is executing a preemptible task r:
6.                  choose the ordering of t and r on c that results in the highest net Util;
7.                  choose the (t,r) ordering among all cores
                    that has the overall maximum net Util;
8.                  find the idle core that maximizes Util for t;
9.                  if the net Util of the (t,r) ordering on the core found above is greater than
                    the net utility of t on the idle core and r on its current core:
10.                 then
11.                     choose the (t,r) ordering;
12.                 else
13.                     choose the idle core for t;
14.         else
15.             find idle cores that maximizes Util for t;
16.     select task t from mappable tasks with task/core
        combination that has the highest maximum Util for t;
17.     if t is part of a (t,r) pair and r should execute first:
18.     then
19.         do nothing with t;
20.     else
21.         assign t to that core (preempting if needed);
22.     remove t from mappable tasks;
23. end while
```

and c2. A task t1 is currently executing on core c1 and will earn 2.0 Util if it finishes its execution. The resource manager is choosing an assignment for a task t2, which can either (a) preempt task t1 and start executing on core c1 where it will earn 2.5 Util or (b) start executing on an idle core c2 where it will earn 1.0 Util (It will take longer for t2 to complete on c2 than c1). The Max Util Preempt Diff heuristic will choose to start executing the task on c2 because the Utility Difference is larger for that core (1.0 instead of 0.5 for c1). The Max Util Preempt Greedy heuristic would have chosen to assign task t2 to core c1 (preempting task t1) because it will get more utility for t2.

4.3.5.4. Maximum Util Pair Preemption

The Maximum Util Pair preemption technique (Max Util Preempt Pair) tries to greedily maximize the net Util earned by two tasks at once to choose which task should be assigned next. This technique is shown in Algorithm 7. The motivation for this heuristic is to consider the amount of utility that would be lost in the currently executing task if a preemption were to occur.

In the first phase of this heuristic there are two cases. The first is used for all tasks that can preempt other tasks. In this case (lines 3-13 of Algorithm 7), two possible orderings of the tasks are considered for each core executing a preemptible task. The first ordering is where the preempting task preempts the executing task, finishes its execution, and then the preempted task resumes and finishes its execution on the same core. (The preempted may not actually resume on this core; this rule is just to guide the heuristic.) The second ordering is where no preemption occurs and the executing task finishes its execution before the task that is being considered is assigned to the core and finishes its own execution. If the second ordering results in higher Util, preemption is not considered for that core. For each core, the ordering with the highest combined Util is chosen. Among all cores, the ordering with the highest overall combined Util is selected to be compared with running the task on idle cores.

Next, the heuristic considers the idle cores by finding the idle core that has the highest Util for the task. The heuristic considers pairing the task executing on this idle core with its best pair that was found when considering preemption. The net Util of this pair would be the Util of the task being considered on the idle core plus the Util of the currently executing task on some core in the system (the best pair found above). This process of considering an idle core is shown on lines 7-10 of Algorithm 7. This is done to ensure that idle cores are treated fairly alongside cores with preemptible tasks.

Tasks that cannot preempt other others use the same method for choosing an idle core as the Max Util heuristic (without preemption). This is shown on lines 14-15 of Algorithm 7. During the second phase of this heuristic (shown on lines 16-22 of Algorithm 7), the task with the overall maximum Util for its allocation is selected. Only the Util of the task being assigned is considered. Otherwise, tasks that cannot preempt other tasks would be at a disadvantage to those that can preempt other tasks. This is because tasks that cannot preempt other tasks would not be paired with any task in the first phase. If the task that is selected in the second phase chose a pair during the first phase where it is the second task to be executed on some core, then do nothing with it during this mapping event (it will be considered in the next mapping event). Otherwise, the task is assigned to the core and is removed from the set of mappable tasks, and the preempted task is added to the set of mappable tasks. This heuristic repeats until the set of mappable tasks is empty or there are no more valid assignments for tasks in the set of mappable tasks.

4.3.6. Heuristics with Utility-per-Time

All of the heuristics described in this section used Util as an objective function. These heuristics can be modified to utilize UPT by replacing all instances of Util in the heuristic descriptions with UPT. A common concern with preemption is that it becomes difficult to execute long running jobs

because they execute during a greater number of mapping events creating more chances for the task to be preempted. When used with Max UPT, our preemption techniques alleviate this issue because the calculated UPT values are larger for tasks that have already started executing. This is because as a task executes, its remaining execution time will decrease, while the utility that it will earn remains constant. This results in a situation where preemption of tasks that have already completed most of their execution is less likely than when the task first started executing in the system. Preemption of long running jobs is thus unlikely unless a task of high importance arrives in the system. In addition, using UPT increases that chance that a preempted task will be quickly resumed once there are opportunities to continue its execution.

4.4. Simulation Setup

4.4.1. Overview

The environment parameters detailed in this section were selected based on discussions with researchers from ORNL and DoD. Some of the parameters described in this section are varied in Section 4.5. Our simulations of this environment take place over 28 simulated hours. The first four hours populate the idle system with tasks. This allows data for the subsequent 24 hours of execution to be collected with the system in a steady state of execution. For each environment, we generate 64 simulation trials using the procedure described in the rest of Section 4.4.

4.4.2. System Generation

The cluster environment is constructed from five heterogeneous clusters with an average of 160 cores each. Although this is a small system relative to many HPC systems today, it allows us to easily experiment with a large variety of workloads.

4.4.3. Workload Generation

In this environment, tasks include two main types: critical tasks and non-critical tasks. Critical tasks have a starting utility of eight and non-critical tasks have a starting utility of one. For the experimental results shown in this paper, 20% of tasks are critical tasks and 80% of the tasks are non-critical tasks. In one of the experiments shown in Section 4.5, the starting utility values of critical tasks were also varied.

Each task type has an associated execution time. If the task type is used for critical tasks, each task has an average execution time of ten minutes. Otherwise, the task type represents a non-critical task and has an average execution time of 50 minutes. The execution time of each individual task type on one of the clusters is determined by sampling a gamma distribution with a coefficient of variation (COV) of 0.1. In our experiments, the execution time of all task types was varied.

The heterogeneity over the three clusters is defined using the method from [2] with a COV parameter of 0.3. Specifically, this means that the execution time generated above for one of the clusters is used as the mean for another gamma distribution. This gamma distribution has a COV of 0.3 and is sampled to get the execution time of the task on the other two clusters. These execution times are then used to populate the ETC matrix that is used by the resource manager.

The size of the workload for this system is determined experimentally to ensure that the system remains oversubscribed during the 24 hours of its steady state execution. To achieve this, an average of 75 tasks need to arrive for each core in the system. For this system, this requires an average of 60,000 tasks arriving over a 24 hour period.

For each task, a utility function is generated. In this study, we consider a variety of utility function classes. One of these classes models utility functions using step functions. For tasks with the highest urgency, this step function has a width equal to the average execution time of the task

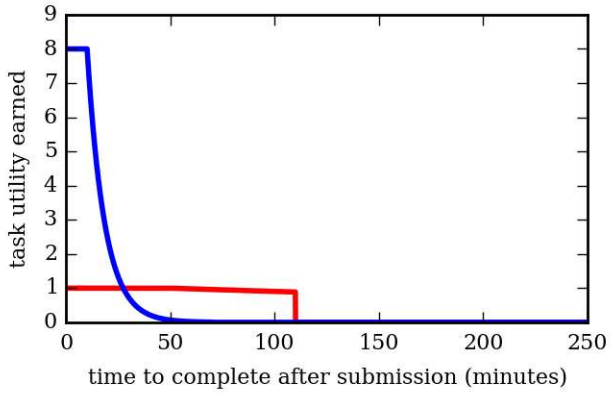


Figure 35. Two utility functions generated using the decaying utility class described in Section 4.4.3. The blue function is generated for critical tasks and the red function is generated for non-critical tasks.

where the task will earn its starting utility. After this period, the utility function drops to zero. Non-critical tasks have a constant interval width equal to ten times their mean execution time and then drop to zero immediately after this interval. We also experimented with the set of 20 utility classes similar to those in Figure 34 from our past work in [15]. In addition, we defined a "decaying utility class" where all tasks have a constant interval equal to their average execution time at the beginning of the utility function and the rest of the utility function is a single period of decay defined using the model in [15] with an urgency parameter of 0.3 and a length of 200 minutes. The utility functions obtained from this utility class are shown for critical tasks in blue and non-critical tasks in red in Figure 35.

Tasks in the workloads of this study are assumed to arrive in bursts. Each burst consists of a number of tasks of the same task type with identical utility functions that arrive at the same time. We define the burst size of the system as the average number of tasks that arrive in each burst. For most of our experiments, the burst size parameter is 64, but this is varied in some experiments. The exact number of tasks arriving in each burst can vary by up 50% of the mean, i.e., for our average burst size of 64 tasks the number of tasks arriving in any burst is between 32 and 96,

inclusive. This value is determined by sampling an integer from a uniform distribution. Bursts of the same task type are spread out throughout the day using a sinusoidal arrival rate (the number of bursts arriving for a task type will be more frequent during some periods of the day than others). The actual distribution of a task type's burst arrival times are randomly generated using a Poisson process from the task's arrival rate.

The majority of the results shown in Section 4.5 assume that all tasks are preemptible and all tasks can be preempted. We will also show one set of results in Section 4.5 where we experimented with these preemption flags.

4.5. Results

All results shown are averaged over 64 simulation trials and are shown with 95% mean confidence intervals. In each trial, the environment is generated as described in Section 4.4. Because of the presence of randomness in some parts of the system, the exact values of many characteristics of the environment vary between simulation trials (e.g., the number of tasks arriving and the execution time of each task type). We define the maximum system utility as the utility earned if all tasks started executing immediately upon arrival in their fastest cluster and earned utility equal to their starting utility. In most cases, this maximum utility is unobtainable because the system is oversubscribed. In all of the results, the execution time of the best performing heuristics is insignificant in comparison to the simulated execution times of the tasks and the mapping event interval of one minute. In our simulations, all of the heuristics except for Max Util Preempt Pair and Max UPT Preempt Pair took approximately seven seconds to execute each mapping event in the worst case. The Max Util Preempt Pair and Max UPT Preempt Pair heuristics took over a minute to execute in the worst case, meaning that these preemption heuristics would not be feasible to use in a system with this size. However, the best performing heuristics could be

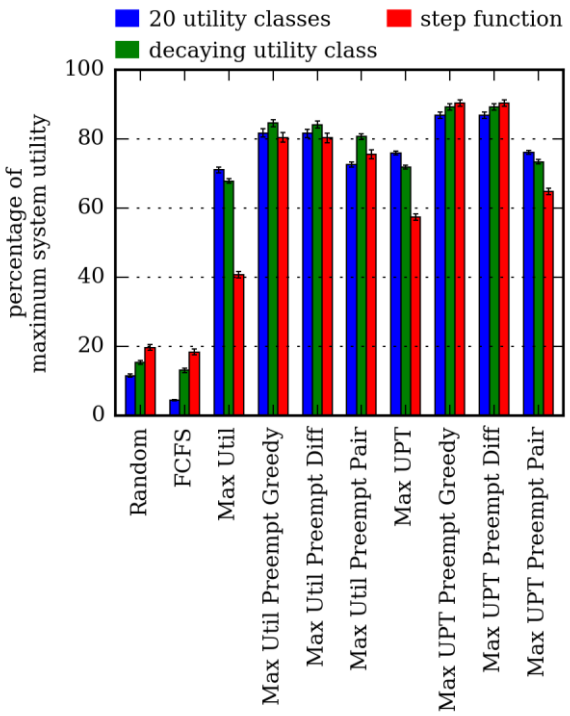


Figure 36. The percentage of maximum system utility is shown for workloads where the utility class used to determine utility functions is varied from step functions, a single decaying utility class, and 20 utility classes as described in Subsection 4.4.3. The results are shown with 95% mean confidence intervals.

implemented in a real system with a mapping event interval of one minute without delaying the assignment of tasks during each mapping event.

In Figure 36, the percentage of maximum system utility is shown for workloads where the type of utility function is varied between step functions, a single decaying utility class, and 20 different utility classes. These types of utility functions were described in Subsection 4.4.3. These results show that utility-based heuristics outperform Random and FCFS. The preemption-capable heuristics always improve upon the utility earned by Max Util and Max UPT. This is because the preemption-capable heuristics are able to earn utility from the critical tasks even when the utility function of the critical task require that they start executing almost immediately upon arrival. This is especially important in the step function case.

In Figure 37, the percentage of maximum system utility is shown for workloads that have burst sizes of 1, 16, 32, 64, and 128. When the burst size is 1, there are no bursts of tasks arriving in the system. In the Figure 37 results, the 20 utility classes described in Subsection 4.4.3 are used. These results show that as the burst size of the workload increases, Max Util and Max UPT see a significant decrease in performance. This is because it becomes more difficult to assign the critical tasks as they arrive because a large group of critical tasks will arrive while only some cores in the system are idle or will become idle soon. The Max Util Preempt Greedy, Max UPT Preempt Greedy, and Max Util Preempt Diff, and Max UPT Preempt Diff heuristics see a much smaller decrease in performance as the burst size increases because they can preempt non-critical tasks to immediately start executing the newly arrived burst of critical tasks. In the case where the burst size is 128, the best performing preemption techniques are able to improve the utility earned by Max UPT by up to 20%.

In Figure 38, the percentage of tasks that can preempt and can be preempted is varied over 0%, 20%, 40%, 60%, 80%, and 100%. When each task is generated, both of the preemption flags are set to true with a probability equal to that percentage. This probability is considered separately for the two preemption flags (e.g., a task may be able to preempt other tasks but it may not be preemptible by other tasks). When these percentages are 0%, the preemption-capable heuristics are identical to their counterparts that are not preemption-capable. In the Figure 38 results, the 20 utility classes described in Subsection 4.4.3 are used. The heuristics that are not preemption-capable do not change over these different workloads because the preemption flags do not affect their execution. As the percentage of tasks that can preempt and can be preempted increases, the performance of most of the preemption-capable heuristics improves linearly with the percentage. The Max UPT Preempt Pair heuristic does not improve linearly because the UPT of a task that has

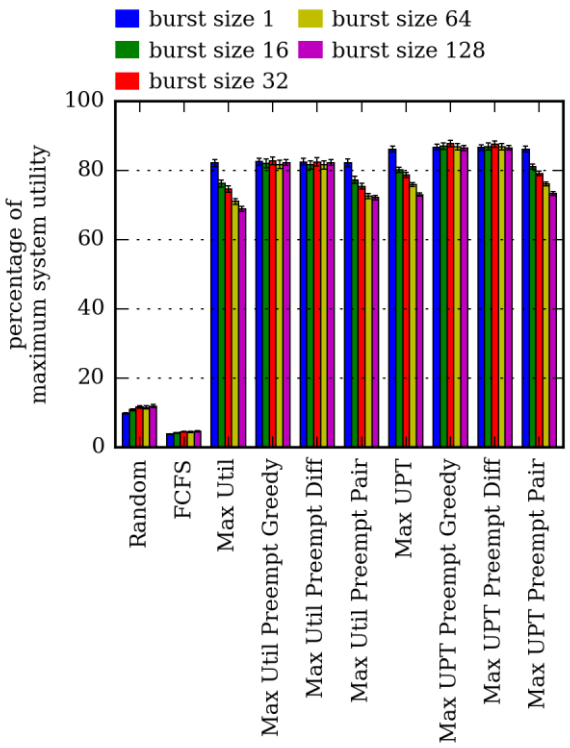


Figure 37. The percentage of maximum system utility earned by each of the heuristics for five different workloads where the burst sizes of tasks arriving are 1, 16, 32, 64, and 128. The results are shown with 95% mean confidence intervals.

almost finished its execution becomes high relative to the other tasks. This high UPT can result in a preemption of that task having a large net UPT value, which results in the Max UPT Preempt Pair heuristic preempting these tasks more frequently than is ideal. These results show that enabling both preemption flags for all tasks does not result in a decrease in overall system performance.

In Figure 39, the percentage of maximum system utility is shown for workloads that have an average execution time for critical tasks equal to 10, 30, and 50 minutes. In the Figure 39 results, the 20 utility classes described in Subsection 4.4.3 are used. As the execution time of critical tasks becomes longer, the system will become more oversubscribed because the average execution time of all tasks has increased. Because of this, all of the heuristics have a slight decrease in their

performance because they cannot execute as many tasks due to the increased oversubscription. In addition, it can be seen that the best preemption-capable heuristics earn more utility than the other heuristics in all cases.

In Figure 40, the percentage of tasks that are completed by each of the heuristics is shown for the same workloads in Figure 39 where the average execution time for critical tasks is equal to 10, 30, and 50 minutes. This is not our performance measure, but this shows that in addition to earning more utility, the utility-based heuristics are also able to complete more tasks. In addition, the Max UPT heuristics are able to complete more tasks than the Max Util heuristics because the Max UPT heuristics are better able to consider the heterogeneity of the system because they consider task execution time when making scheduling decisions. Given a set of tasks with the same utility function, the Max UPT heuristic would be able to assign each of those tasks to the cluster able to execute it in the least amount of time.

Across all of the results in Figure 36, Figure 37, Figure 38, and Figure 39, the percentage of maximum system utility is shown for a variety of workloads. These results show that utility-based heuristics are more effective than the comparison heuristics at maximizing utility. In addition, it can be seen that the Max UPT heuristic (regardless of the preemption technique used with it) outperforms the Max Util heuristic in all of these environments for the reasons described above.

When comparing the preemption techniques, it can be seen that the Max Util Preempt Greedy, Max UPT Preempt Greedy, and Max Util Preempt Diff, and Max UPT Preempt Diff heuristics tend to have close to identical performance because in most instances they make the same allocation decisions. The Max Util Preempt Pair and Max UPT Preempt Pair heuristics perform worse than the other preemption techniques in all of these environments. This is likely because this preemption technique attempts to make decisions about what should occur at future mapping

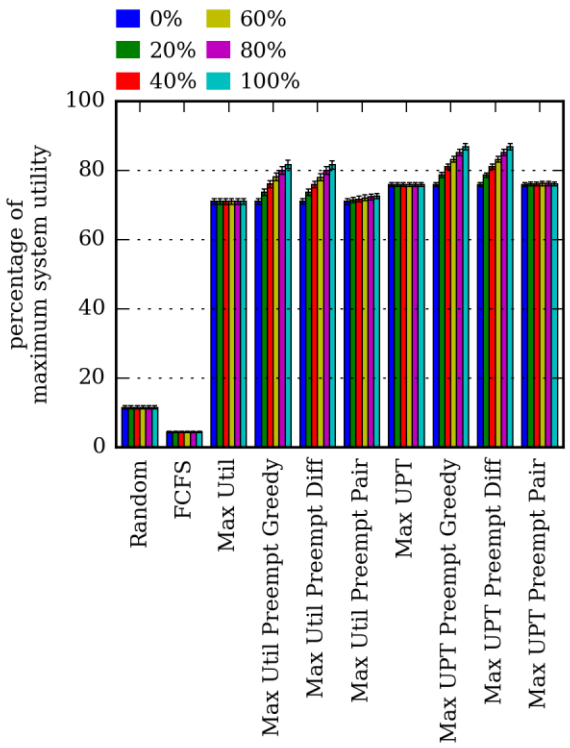


Figure 38. The percentage of maximum system utility earned by each of the heuristics for six different workloads where the percentage of tasks that can preempt and the percentage of tasks that can be preempted are varied over 0%, 20%, 40%, 60%, 80%, and 100%. The results are shown with 95% mean confidence intervals.

events. Because the tasks in this system arrive dynamically, these decisions made by the heuristic may not be carried out when new potentially high utility tasks arrive before the next mapping event. It can also be seen that the Max Util Preempt Pair heuristic performs better relative to the other Max Util heuristics than Max UPT Preempt Pair does relative to the other Max UPT heuristics. This is because the UPT of a task that has almost finished its execution becomes high relative to the other tasks, which can result in this heuristic preempting these tasks more frequently than is ideal.

The Max Preemption and Max Difference Preemption heuristics are the best performing heuristics overall and are able to significantly improve the utility earned by our utility-aware heuristics in many environments. Further, they never result in reduced utility earned relative to

any of the other heuristics. These preemption techniques also have an insignificant execution time overhead that is comparable on these systems to the overhead of running any of the utility-aware heuristics without preemption.

4.6. Related Work

Preemption in scheduling is commonly seen in the literature, but it is rarely the main focus of the research it appears in and there is often limited analysis of it. One area with a considerable amount of research on preemption is scheduling for systems that are running the tasks of MapReduce applications. For example, the research in [6] proposes a new strategy for using preemption to quickly execute high priority jobs. In that study, the proposed scheduler (Global Preemption) attempts to improve the performance of the system by attempting to avoid undesirable preemptions (e.g., avoid preempting jobs that have already been executing for a long period of time). This is similar to the effect that is provided by our Max UPT preemption heuristics, but this work does not consider utility, considers only a homogeneous system (unlike our heterogeneous environment), and considers a set of tasks that is very different from ours in terms of arrival pattern and execution characteristics.

In [7], a small homogeneous parallel environment with a single 32 node cluster is studied. Tasks in this study have simple linearly decaying "value functions," which fit our definition of a utility function. This study had a similar distribution of tasks to ours where 80% of the tasks were "low value" and 20% of the tasks were "high value" with the "high value" tasks having an average of 100 times the importance relative to a "low value" task. The simple preemption technique applied in this study would make a preemption whenever swapping a running job with the first task in the queue would increase the total value earned. In addition, there was a constraint that each task could be preempted at most once. In the best case (i.e., when tasks were had utility

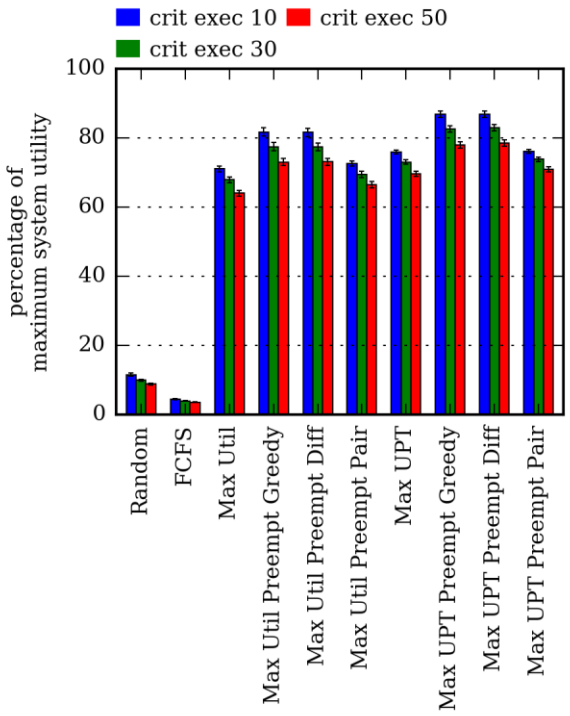


Figure 39. The percentage of maximum system utility earned by each of the heuristics for three different workloads where the average execution times of critical tasks are 10, 30, and 50 minutes. Recall that the execution time of non-critical tasks is 50 minutes. The results are shown with 95% mean confidence intervals.

functions that decayed to zero the fastest) this resulted in an improvement of around 20% in terms of value earned. Our work differs from [7] because that study did not consider heterogeneity and used only one type of utility function.

The authors of [14] propose a preemption-aware heuristic called “Selective Preemption” for scheduling parallel jobs. They considered workload traces from real systems and showed that their technique outperformed some existing scheduling techniques because it was able to obtain good performance for all tasks regardless of their resource requirements. In comparison, the existing techniques either had trouble executing large tasks (i.e., tasks with long execution times that are parallelized across many cores) or small tasks (i.e., tasks with short execution times parallelized over only a few cores) effectively. This work differs from ours because it studies a parallel environment, and uses metrics such as turnaround time and slowdown to measure performance. In

our environments, the metric of performance is utility and it is unimportant if a certain class of tasks has trouble executing as long as the overall utility earned is maximized.

Scheduling is studied for an environment with value functions that can decay to negative values (this is a penalty for failing to schedule a task) in [12]. Here, a significant difference from our work is that the scheduler only selects (and is only penalized for) tasks that it decides to accept (the purpose of penalties is to encourage that the tasks get executed by the site that accepts them). The authors show that their scheduling technique outperforms several existing strategies for maximizing value earned in such an environment.

Utility has been used as a performance measure in a variety of serial and parallel environments, (e.g., [13, 15, 20, 21, 22, 28]). Our work differs from these because none of them consider the possibility of preemption to improve system performance. In addition, [13, 20, 22, 28] do not consider heterogeneity.

In addition to simple heuristics that can be used to quickly make scheduling systems, it is common to study the use of more time consuming techniques such as genetic algorithms. Given enough time, a genetic algorithm can find very good solutions to scheduling problems. In a parallel environment, a genetic algorithm was applied in [20] and was able to outperform common parallel scheduling techniques such as EASY Backfilling. Unfortunately, this genetic algorithm had an average execution time of 8,900 seconds. In an environment like the one studied in our work, where mapping decisions must be made every minute, it is not possible to use a resource manager with that level of execution overhead.

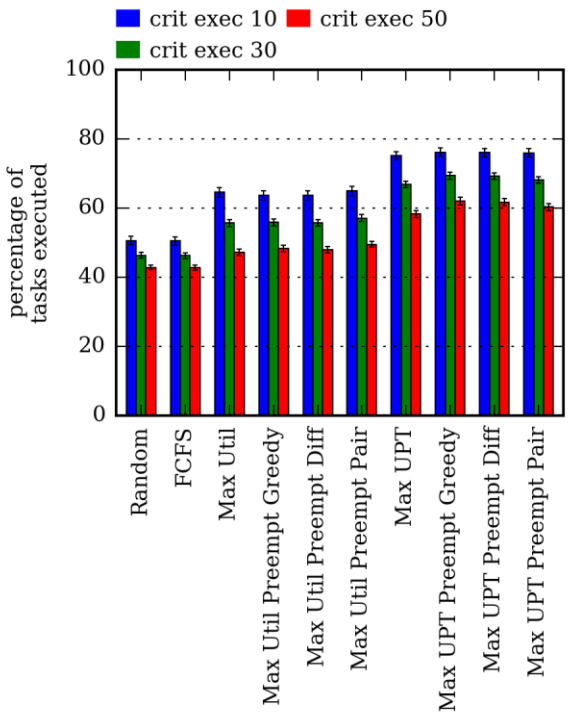


Figure 40. The percentage of tasks completed by each of the heuristics for three different workloads where the average execution times of critical tasks are 10, 30, and 50 minutes. Recall that the execution time of non-critical tasks is 50 minutes. The results are shown with 95% mean confidence intervals.

4.7. Conclusions and Future Work

We designed and evaluated six different preemption-capable heuristics. In environments where preemption is possible, we have shown that these heuristics are able to significantly outperform Random and the common FCFS scheduling technique. In addition, the preemption-capable heuristics were able to outperform the utility-aware heuristics without preemption (Max Util and Max UPT). Detailed analyses of the differences in performance were described for a variety of environments.

An important and significant extension of this work would be to consider the preemption of parallel tasks executing on oversubscribed HPC systems. Our previous work has shown that utility-aware heuristics are still very effective in parallel environments [21, 22]. Because parallel tasks

are common in most HPC systems, designing a resource manager that is capable of efficiently scheduling parallel tasks is an important problem for the HPC community to consider. An environment with parallel tasks would add additional challenges to this problem because it would no longer be realistic to consider the overhead of suspending or resuming a task to be negligible. In addition, it would be much more computationally expensive to consider every possible preemption for each task against different sets of smaller tasks. Thus, different, more complex, and more time-consuming heuristics will need to be designed and analyzed.

Another possible extension to this work is the consideration of energy. Because of the growing need for energy efficient HPC systems, it is important that any scheduler be aware of system energy use resulting from its scheduling decisions.

References

[1] S. Ali, T. D. Braun, H. J. Siegel, A. A. Maciejewski, N. Beck, L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, "Characterizing Resource Allocation Heuristics for Heterogeneous Computing Systems," *Advances in Computers* vol. 63, pp. 91–128, 2005.

[2] S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, and S. Ali, "Representing Task and Machine Heterogeneities for Heterogeneous Computing Systems," *Tamkang Journal of Science and Engineering*, Special Tamkang University 50th Anniversary Issue, vol. 3, no. 3, pp. 195–207, 2000.

[3] S. Arunagiri, M. R. Varela, R. A. Oldfield, R. Riesen, S. Seelam, P. C. Roth, and P. J. Teller, "Modeling the Impact of Checkpoints on Next-Generation Systems," *IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies*, Sep. 2007.

[4] H. Barada, S. M. Sait, and N. Baig, "Task Matching and Scheduling in Heterogeneous Systems Using Simulated Evolution," *10th IEEE Heterogeneous Computing Workshop (HCW 2001)*, pp. 875–882, Apr. 2001.

[5] T. D. Braun, H. J. Siegel, N. Beck, L. Boloni, R. F. Freund, D. Hensgen, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, 2001.

[6] L. Cheng, Q. Zhang, and R. Boutaba, "Mitigating the Negative Impact of Preemption on Heterogeneous MapReduce Workloads," *7th International Conference on Network and Service Management (CNSM)*, Oct. 2011.

[7] B. N. Chun and D. E. Culler, "User-Centric Performance Analysis of Market-Based Cluster Batch Schedulers," *2nd IEEE International Symposium on Cluster Computing and the Grid*, May 2002.

[8] N. Fallenbeck, H.-J. Picht, M. Smith, and B. Freisleben, "Xen and the Art of Cluster Scheduling," *2nd International Workshop on Virtualization Technology in Distributed Computing (VTDC '06)*, 2006.

[9] R. F. Freund and H. J. Siegel, "Heterogeneous Processing," *IEEE Computer*, vol. 26, no. 6, pp. 13–17, June 1993.

[10] A. Ghafoor and J. Yang, "A Distributed Heterogeneous Supercomputing Management System," *IEEE Computer*, vol. 26, no. 6, pp. 78–86, June 1993.

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

[12] D. E. Irwin, L. E. Grit, and J. S. Chase, "Balancing Risk and Reward in a Market-Based Task Service,"*13th IEEE International Symposium on High performance Distributed Computing (HPDC 2004)*, June 2004.

[13] E. Jensen, C. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Systems," *IEEE Real-Time Systems Symposium*, pp. 112–122, Dec. 1985.

[14] R. Kettimuthu, V. Subramani, and S. Srinivasan, "Selective Preemption Strategies for Parallel Job Scheduling," *International Conference on Parallel Processing*, Aug. 2002.

[15] B. Khemka, R. Friese, L. D. Briceño, H. J. Siegel, A. A. Maciejewski, G. A. Koenig, C. Groer, G. Okonski, M. M. Hilton, R. Rambharos, and S. Poole, "Utility Functions and Resource Management in an Oversubscribed Heterogeneous Computing Environment," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2394–2407, Aug. 2015.

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

[17] D. Klusaccek and H. Rudova, "Performance and Fairness for Users in Parallel Job Scheduling," *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, vol. 7698, pp. 235–252. 2012.

[18] C. Leangsuksun, J. Potter, and S. Scott, "Dynamic Task Mapping Algorithms for a Distributed Heterogeneous Computing Environment," *4th IEEE Heterogeneous Computing Workshop (HCW 1995)*, pp. 30–34, Apr. 1995.

[19] D. A. Lifka. "The ANL/IBM SP Scheduling Systems," *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, vol. 949, pp. 295–303, 1995.

[20] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Realtime Operating Systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 613–629, Sep. 2004.

[21] D. Machovec, B. Khemka, S. Pasricha, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, M. Wright, M. Hilton, R. Rambharos, and N. Imam, "Dynamic Resource Management for Parallel Tasks in an Oversubscribed Energy-Constrained Heterogeneous Environment," *25th Heterogeneity in Computing Workshop (HCW 2016)*, May 2016.

[22] D. Machovec, C. Tunc, N. Kumbhare, B. Khemka, A. Akoglu, S. Hariri, and H. J. Siegel, "Value-Based Resource Management in High-Performance Computing Systems," *7th Workshop on Scientific Cloud Computing (ScienceCloud 2016)*, May/June 2016.

[23] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 107–131, 1999.

[24] M. Maheswaran, T. D. Braun, and H. J. Siegel, "Heterogeneous Distributed Computing," in J. G.Webster, editor, *Encyclopedia of Electrical and Electronics Engineering*, vol. 8, pp. 679–690. JohnWiley, New York, NY, 1999.

[25] A. Mishra, S. Mishra, and D. S. Kushwaha, "An Improved Backfilling Algorithm: SJF-B," *International Journal on Recent Trends in Engineering & Technology*, vol. 5, no. 1, pp.78–81, Mar. 2011.

[26] A. W. Mu'alem and D. G. Feitelson, "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 529–543, June 2001.

[27] H. Singh and A. Youssef, "Mapping and Scheduling Heterogeneous Task Graphs Using Genetic Algorithms," in *5th IEEE Heterogeneous Computing Workshop (HCW 1996)*, pp. 86-97, Apr. 1996.

[28] D. Vengerov, L. Mastroleon, D. Murphy, and N. Bambos, "Adaptive Data-Aware Utility-Based Scheduling in Resource-Constrained Systems," Technical Report 2007-164, Sun Microsystems, Inc., 2007.

[29] D. Xu, K. Nahrstedt, and D. Wichadakul, "QoS and Contention-Aware Multi-Resource Reservation," *Cluster Computing*, vol. 4 no. 2, pp. 95–107, Apr. 2001.
[30] J. Yang, A. Khokhar, S. Sheikh, and A. Ghafoor, "Estimating Execution Time for Parallel Tasks in Heterogeneous Processing (HP) Environment," *Heterogeneous Computing Workshop (HCW)*, pp. 23–28, Apr. 1994.
[31] D. Machovec, S. Pasricha, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, M. Wright, M. Hilton, R. Rambharos, T. Naughton, and N. Imam, "Preemptive Resource Management for Dynamically Arriving Tasks in an Oversubscribed Heterogeneous Computing System," *26th Heterogeneity in Computing Workshop (HCW 2017)*, 2017 International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2017), pp. 54–64, Orlando, FL, May 2017.

Chapter 5

Surveillance Mission Scheduling with Unmanned Aerial Vehicles in Dynamic Heterogeneous Environments[4]

5.1. Introduction

Unmanned aerial vehicles (UAVs) are used in many environments to gather information, such as in active battlefields scenarios. An example of such a scenario is shown in Figure 41. In this example, seven UAVs are being used to gather information about nine targets. We assume that a UAV can only surveil a single target at a time, so this is an *oversubscribed* scenario because there are more targets than UAVs and it will not be possible to surveil all targets simultaneously with the fleet of UAVs. To gather as much useful information about the targets as possible, it is necessary to conduct mission planning and scheduling to determine how the UAV fleet should effectively surveil the targets.

As both the number of UAVs that are active simultaneously and the number of targets available in an environment increases, it becomes necessary to reduce the amount of human control and human scheduling required to operate them effectively [1]. This can be accomplished by designing and deploying heuristic techniques that can find effective mission scheduling solutions.

In this study, our focus is on the design of mission scheduling techniques capable of working in dynamic environments that are suitable for determining effective mission schedules in real-time.

---

[4] This work was done jointly with former Ph.D. student Ryan Friese. The full list of co-authors for this work is at [24]. Preliminary versions of this work appeared in [21, 22, 23].

Because scheduling problems of this type are, in general, known to be NP-Hard, finding optimal solutions is not feasible [2]. Due to this, we consider fast heuristic techniques to find mission schedules and evaluate their performance compared to other heuristics. These techniques find mission schedules for our scenarios in less than a second on average.

To effectively compare and evaluate these techniques, we measure system-wide performance using a new metric called *surveillance value*. Surveillance value measures the overall performance of all information gathered by the UAVs, based on parameters such as the number of surveils that occur, the quality of information gathered by each surveil (e.g., image resolution), the importance of each target, and the relevance of the information obtained for a specific target. The novel contributions in this work include:

- the design of new mission scheduling heuristics that are used to dynamically determine which UAVs and sensors should be used to surveil each target;
- the modification of the heuristics using preemption and filtering techniques that enable more efficient utilization of the UAVs in a fleet;
- the construction of a model for UAVs surveilling targets in energy-constrained, dynamic scenarios where the value of each surveil can be quantified;
- the development of a novel system-wide performance measure of the information gathered by all UAVs considering multiple factors;
- the creation of a model for randomly generating scenarios defined by a set of UAVs and targets for the purpose of evaluating mission scheduling techniques, such as the heuristics considered in this work;
- the analysis and comparison of heuristics across many varied simulated scenarios.

Figure 41. An example scenario with seven UAVs and nine targets. Arrows drawn from a UAV to a target signify that the target is currently under surveillance by the UAV.

This paper is organized as follows. In Section 5.2, the system model and environment are described. The methods used by both the novel mission scheduling heuristics and the comparison heuristics evaluated in this study are presented in Section 5.3. Section 5.4 contains the specific process used to generate the scenarios we use in our simulations. In Section 5.5, we show the results of the simulations and use the results to analyze and compare the behavior and performance of the heuristics. Related work is discussed in Section 5.6 and finally in Section 5.7 we conclude and discuss possible future work.

5.2. System Model

5.2.1. Overview

The system considered in this study consists of a heterogeneous set of UAVs (with varying sensor and energy characteristics) and a heterogeneous set of targets (with varying surveillance requirements). These sets of UAVs and targets are dynamic, meaning that UAVs and targets can be added or removed from the sets at any time. Additionally, specific characteristics of the UAVs

and targets can change dynamically at any time. We make a simplifying assumption that every UAV is always close enough to every target and has an unobstructed view of every target so that any sensors available to a UAV can surveil any target at any time.

Because UAVs cannot stay airborne indefinitely, this work considers mission scheduling strategies for a single day. At the end of the day, all UAVs would be able to return to their base of operations to refuel or recharge. In this study, a UAV can only surveil a single target at any given time. The problem space we explore primarily consists of oversubscribed systems, which means that there are fewer UAVs than targets. This will prevent all available targets from being surveilled simultaneously. While the majority of the environments in this study are oversubscribed, the techniques we design and evaluate are still applicable to undersubscribed systems.

Because UAVs cannot stay airborne indefinitely, this work considers mission scheduling strategies for a single day. At the end of the day, all UAVs would be able to return to their base of operations to refuel or recharge. The energy needed for the UAV's fixed flight plan is not included in the energy available to the sensors in our model. In this study, a UAV can only surveil a single target at any given time. The problem space we explore primarily consists of oversubscribed systems, which means that there are fewer UAVs than targets. This will prevent all available targets from being surveilled simultaneously and 100% of desired surveils will not occur. While the majority of the environments in this study are oversubscribed, the techniques we design and evaluate are still applicable to undersubscribed systems.

5.2.2. Target and UAV Characteristics

In our environment, each UAV has a single energy source with a fixed amount of *total energy* available to it. In our subsequent discussions, we normalize this value so that the maximum amount of energy available to any UAV is less than or equal to 1.0. Every UAV is equipped with one or

more sensors that can be used to surveil targets. The *sensor types* considered in this work are visible light sensors (VIS), infrared light sensors (IR), synthetic-aperture radars (SAR), and light detection and ranging systems (LIDAR). Each UAV cannot have more than one sensor of a given type, which is a simplifying assumption in this work. The heuristics presented in this study will function in environments with multiple sensors with the same type. Each sensor available to a UAV also has an associated *sensor quality* value ranging from 1 (worst) to 10 (best) and a *rate of energy consumption*, which is normalized to the total energy available to the UAV and ranges from 0.0 to 1.0 normalized units of energy per hour. All sensors available to a UAV use the same energy source. The energy needed for the UAV's fixed flight plan is not included in the energy available to the sensors in our model. An example of UAV characteristics is shown in Table 9 for a fleet of four UAVs.

Targets represent locations of interest to be potentially surveilled by UAVs. A *priority* value is assigned to each target, which represents the overall importance of surveilling the target. Priority values are positive integers between 1 and 10, where higher numbers represent more important targets. Each target has a *surveillance time*, which specifies the number of hours that a UAV should spend surveilling the target in a single surveil. Because the kind of information that is useful for each target may vary, targets have a set of *allowed sensor types*, which constrains which sensors can surveil the target. Each of these allowed sensor types has an associated *sensor affinity* value, which range from 1 (worst) to 10 (best) and measures how useful or relevant the information gained from that sensor type is for the target. Table 10 contains target characteristics for an example set of six targets. Additionally, each target has a set of *surveillance intervals*, representing the time intervals in which a single surveil of the target should occur.

Table 9. UAV characteristics

characteristics	UAV 1	UAV 2	UAV 3	UAV 4
total energy	1.0	0.5	0.8	0.8
sensor type	VIS	SAR | IR	VIS | IR	SAR | LIDAR
energy consumption/hour	0.05	0.15 | 0.08	0.1 | 0.05	0.15 | 0.05
sensor quality	7	7 | 5	9 | 3	7 | 4

Table 10. Target characteristics

characteristics	target 1	target 2	target 3	target 4	target 5	target 6
priority	3	4	5	6	8	10
surveillance time	0.97 hours	3.02 hours	1.77 hours	2.73 hours	1.27 hours	2.52 hours
allowed sensors	SAR	VIS | IR	SAR | IR | LIDAR	VIS | IR | LIDAR	VIS | SAR | IR | LIDAR	VIS | IR
sensor affinity	6	7 | 4	2 | 8 | 5	3 | 1 | 8	9 | 2 | 5 | 4	8 | 6

5.2.3. Dynamic Events

Characteristics of the UAVs and targets in a scenario can change dynamically during the day. For example, changes in the weather may affect the quality of information collected by certain sensor types, which can be modeled in this study by a change in the sensor affinity value for targets affected by this change in weather. We model this as a change in sensor affinity because the weather would be local to one or more targets and would affect the sensors of all UAVs surveilling that target.

The dynamic changes that we model for UAVs are: (a) adding and removing UAVs from the scenario, (b) removing sensor types from UAVs, and (c) modifying the sensor quality for sensors of the UAVs. The set of targets can also dynamically change: (a) new targets can be added or removed from the scenario, (b) priority of targets can be adjusted, (c) time that a target should be surveilled in a single surveil can be modified, (d) allowed sensor types can be added or removed from the target, and (e) sensor affinities for each allowed sensor type can be altered.

In this study, we assume that any dynamic changes are unexpected and that the techniques that assign UAVs to surveil targets have no information about (a) when the changes will happen, (b)

which UAVs and targets will have their characteristics changed, and (c) which specific characteristics will be changed. Additionally, the specific intervals of time during the day when each target can be surveilled are not known in advance.

f the UAVs and targets in a scenario can change dynamically during the day. For example, changes in the weather may affect the quality of information collected by certain sensor types, which can be modeled in this study by a change in the sensor affinity of targets affected by this change in weather.

The dynamic changes that we model for UAVs are: (a) adding and removing UAVs from the scenario, (b) removing sensor types from UAVs, and (c) modifying the sensor quality for sensors of the UAVs. The set of targets can also dynamically change: (a) new targets can be added or removed from the scenario, (b) priority of targets can be adjusted, (c) time that a target should be surveilled in a single surveil can be modified, (d) allowed sensor types can be added or removed from the target, and (e) sensor affinities for each allowed sensor type can be altered.

In this study, we assume that any dynamic changes are unexpected and that the techniques that assign UAVs to surveil targets have no information about (a) when the changes will happen, (b) which UAVs and targets will have their characteristics changed, and (c) which specific characteristics will be changed. Additionally, the specific intervals of time during the day when each target can be surveilled are not known in advance.

5.2.4. Surveillance Value

To evaluate the performance of different techniques for assigning UAVs to targets, it is necessary to measure the worth of individual surveils by a UAV on a target. For a UAV (u), target (t), and used sensor type (s) the *value* of a surveil ($\sigma(u, s, t)$) is given by the product of the priority (ρ), sensor affinity (α), and sensor quality (γ):

$$\text{value}(\sigma(u,s,t)) = \rho(t) * \alpha(t,s) * \gamma(u,s). \quad (1)$$

The total *surveillance value* earned over an interval of time is then defined by the sum of values earned by all surveils performed by UAVs in that interval of time:

$$\text{surveillance value} = \sum_{\sigma \in \text{ surveils performed}} \text{value}(\sigma). \quad (2)$$

If a surveil is not fully completed, then a partial value will be earned for that surveil, which is directly proportional to the fraction of the surveil that was completed. A partial surveil can occur when the target's surveillance interval ends, the UAV runs out of energy, there are dynamic changes in the environment that stop the surveil, or a heuristic preempts the surveil.

In the case where a characteristic of the target or UAV that affects the value of the surveil changes during the surveil, then the value of the surveil before the change is calculated as a partial surveil that ends when the change occurs. The value of the surveil after the change is similarly calculated as a partial surveil using the remaining time until the end of the surveil or until another characteristic that would affect the value changes. For example, if during a five-hour surveil the priority of a target is doubled after three hours, then the value earned for the first three hours would be calculated as 60% of a full surveil using the initial priority and the value earned for the last two hours would be calculated as 40% of a full surveil using the doubled priority.

5.2.5. Problem Statement

The goal of our proposed scheduling heuristics is to maximize surveillance value obtained over a day. This problem is constrained by the total energy available to each UAV. In this study, this constraint is only applied to the energy consumed by a UAV's sensors. Additionally, each UAV can only surveil one target and only operate one of its sensors at any time; similarly, at any point in time, each target can only be surveilled by one UAV. These are simplifying assumptions used in this study.

5.3. Mission Scheduling Techniques

5.3.1. Mapping Events

Mapping UAVs to targets refers to the process of determining which UAVs will surveil which targets, which sensors will be used for surveils, and when the surveils will occur. When preemption is not considered, a UAV is an *available UAV* to be mapped if it is not currently surveilling targets and has energy remaining, and a target is an *available target* to be mapped if it is not currently being surveilled and it is eligible for being surveilled based on its surveillance intervals. A sensor of a UAV is said to be a *valid sensor type* for a given target if that sensor type is also in the target's list of allowed sensor types. If a UAV has a valid sensor type for a target, it is called a *valid UAV* for that target. Only valid UAVs are considered for mapping to a given available target.

The instant when a mapping of available UAVs to available targets occurs is called a *mapping event*. At a mapping event, a mission scheduling technique is used to assign available UAVs to surveil available targets based on the current state of the system. In this study, all techniques presented are real-time heuristics to allow mapping events to be completed in less than a second on average for the problem sizes we consider. There are different techniques for deciding when a mapping event should be initiated, e.g., at fixed time intervals or due to changes in the environment. In this study, we consider the case where mapping events occur with a fixed time interval. Most of our simulations use a fixed time interval of five minutes. We examined the impact of this interval as a part of our simulations and found that other time intervals do not significantly improve performance. In a real-world implementation, this interval of time can be derived based on empirical evaluations of the characteristics of the actual system.

5.3.2. Comparison Techniques

5.3.2.1. Random

At a mapping event, the *Random* technique considers available targets in a random order. For each target, a random available valid UAV and a random valid sensor type of that UAV are selected. The selected UAV and sensor type are assigned to surveil the target. This results in both the target and the UAV becoming unavailable for new assignments until this new surveil completes. If there is no available UAV that has a valid sensor type for the target, then no UAV is assigned to the target. This repeats with the next target in the random ordering until there are no more assignments of UAVs to targets possible in the current mapping event.

5.3.2.2. Random Best Sensor

The *Random Best Sensor* heuristic is similar to the Random technique, except that it uses knowledge about the sensor quality of UAVs and the sensor affinity of targets to make decisions that are likely to result in higher surveillance value. Like the Random heuristic, available targets are considered in a random order and a UAV with a valid sensor type for this target is selected at random. Instead of selecting a random valid sensor type from the UAV, this heuristic chooses the sensor type with the maximum product of the UAV's sensor quality and the target's sensor affinity. Because both values are directly used along with the target's priority in the calculation for the value of a surveil, this strategy will often select higher value surveils compared to the Random heuristic. Next, the same process used by the Random heuristic occurs: the UAV is assigned to surveil the target with this sensor type and the heuristic continues with the next randomly ordered target until no more assignments are possible.

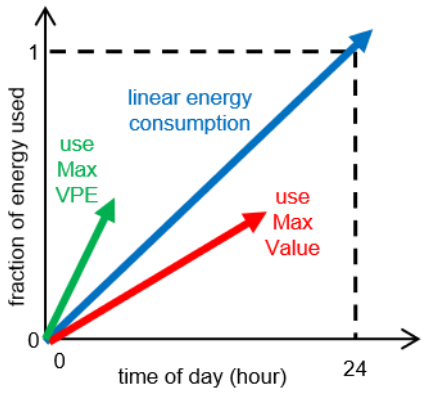


Figure 42. A visualization of the decision-making process employed by the Metaheuristic. If the current energy consumption during the day of a UAV is below the linear energy consumption line, then Max Value is used for the UAV at the current mapping event. Otherwise, Max Value Per Energy (Max VPE) is used.

5.3.3. Value-Based Heuristics

5.3.3.1. Overview

The value-based heuristics in this study are designed to search through valid combinations of UAVs, targets, and sensor types to greedily assign UAVs to surveil targets based on the surveillance value performance measure. A valid combination is represented by an available target, a valid available UAV for that target, and a valid sensor type of the UAV for the target.

5.3.3.2. Max Value

At a mapping event, the *Max Value* heuristic starts by finding a valid combination of a UAV, target, and sensor type that results in the maximum possible value for a single surveil. If there are multiple valid combinations with the same maximum possible value, then one of these combinations is selected arbitrarily. The heuristic then assigns the UAV from the selected combination to surveil the selected target with the selected sensor type. This process of finding the maximum value combination and starting a surveil based on the combination repeats until no more assignments of available UAVs to available targets are possible in the current mapping event.

5.3.3.3. Max Value Per Time

The *Max Value Per Time* heuristic is identical to Max Value except for one difference. Instead of selecting the valid combination that results in the maximum possible value for a surveil, Max Value Per Time instead selects the valid combination that results in the maximum possible value divided by surveillance time of the target (based on a complete surveil of the target).

5.3.3.4. Max Value Per Energy

The *Max Value Per Energy* heuristic is identical to Max Value except that instead of selecting the valid combination that results in the maximum possible value for a surveil, Max Value Per Energy instead selects the valid combination that results in the maximum possible *value per energy (VPE)*, equal to value divided by the projected energy consumed by the UAV for that surveil. The projected energy consumption can be easily calculated from the energy consumption rate of the selected sensor type and the surveillance time of the selected target (based on a complete surveil of the target). We have used the general concept of performance per unit time and performance per unit of energy in prior work in a high-performance computing environment, e.g., [3], [4].

5.3.4. Metaheuristic

The value-based heuristics described in Section 5.3.3 are designed to perform well in specific situations and using the wrong heuristic for a scenario could result in poor performance. Because there may be insufficient information to predict which heuristic should be used, we design a metaheuristic to intelligently combine the best performing value-based heuristics. This does not include the Max Value Per Time heuristic because in the scenarios we consider, Max Value Per Time never performs better than either Max Value or Max Value Per Energy on average. The *Metaheuristic* uses a two-phase process to find good surveillance options. The general strategy employed by this metaheuristic is illustrated in Figure 42. The metaheuristic keeps track of the

historical rate of energy consumption of each UAV and will use either Max Value or Max Value Per Energy depending on whether the historical rate of energy consumption is above or below a linear rate of energy consumption that would result in running out of energy at the end of the 24-hour period we consider.

In the first phase, the Metaheuristic selects a candidate target and valid sensor type for each UAV. The fraction of the day that has passed (δ) and the fraction of the UAV's energy that has been consumed (ε) are used to determine if the strategy used by the Max Value or Max Value per Energy heuristic would be most effective. If $\delta > \varepsilon$, energy is being consumed slowly and Max Value is used. Otherwise, the UAV has been consuming energy at a relatively high rate and the strategy from Max Value Per Energy can be used to make energy-efficient decisions. Based on this choice, either the valid combination using the UAV that results in the maximum possible value or the maximum possible value divided by energy consumed is selected as the best candidate combination for the current UAV. The first phase ends when every UAV has a candidate combination selected. Note that multiple UAVs can select the same target as their candidate.

The second phase is used to determine which UAV from the first phase should be assigned to its candidate target and sensor type. Unlike the first phase, it is unnecessary to use strategies from multiple value-based heuristics in the second phase. This is because energy is a constraint for individual UAVs and not for the overall system. At the system level, all that is relevant to maximizing surveillance value is the value of each surveil. Thus, we choose the UAV with a candidate combination that results in the maximum possible value earned by its corresponding surveil. This chosen UAV is assigned to surveil its target. This process of selecting candidates in the first phase and making an assignment of the best candidate in the second phase is repeated until

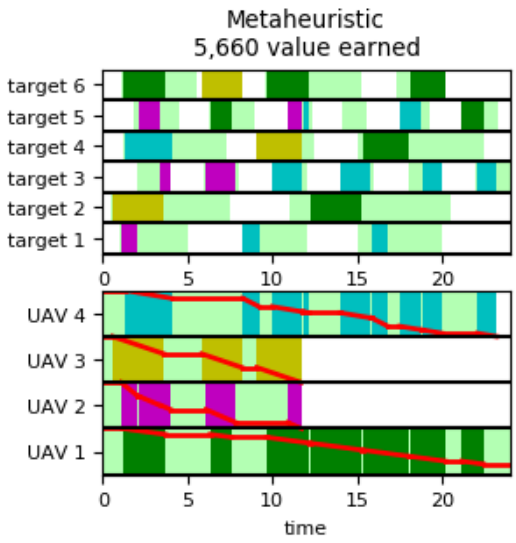


Figure 43. The mapping produced by the Metaheuristic for our example scenario in Table 9 and Table 10. This mapping earns a total value of 5,660 during the 24 hours we consider. The red lines represent the percentage of remaining energy for each UAV. Light green regions represent the surveillance intervals for each target and when a UAV has energy remaining. White regions represent when a UAV or target is not available. Dark green (UAV 1), purple (UAV 2), yellow (UAV 3), and blue (UAV 4) regions represent when a surveil is active using each specific UAV.

no more assignments are possible in the current mapping event. An example of a mapping produced by the Metaheuristic is shown in Figure 43.

5.3.5. Heuristic Modifications

5.3.5.1. Overview

We consider two modifications that can be applied to any of the previously described heuristics. One of these modifications allows the heuristics to preempt surveils and the other limits the options available to a heuristic to improve the chance that the heuristic makes optimal decisions.

5.3.5.2. Preemption

Preemption is a modification to a heuristic that increases the number of choices available. Specifically, in addition to assigning available UAVs to available targets, the heuristic can stop

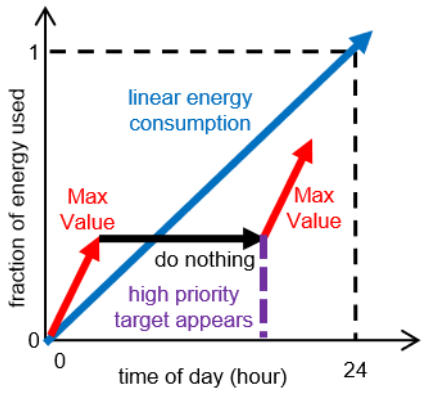


Figure 44. A visualization of the decision-making process employed by the filtering technique. This modification is designed to prevent UAVs from using all of their energy quickly, which is useful when high priority targets would be available late in the day.

any surveil that is currently in progress, which will cause the affected UAV and target to become immediately available. Because we assume that all UAVs are always able to begin surveilling any target immediately (see Subsection 5.2.1), we do not consider any overhead time due to preempting the surveils.

This modification adjusts the greedy heuristics described in this section so that the set of available targets and UAVs includes all targets and UAVs as long as the new combination that will preempt an existing surveil is better in terms of the metric used by the heuristic (e.g., a surveil is better for Max Value if it earns more value).

5.3.5.3. Filtering

Filtering is a modification to a heuristic that reduces the number of choices available. Specifically, this modification is designed to control the rate of energy consumption of each UAV so that it will run out of energy close to the end of the day. An example of the benefits of filtering is shown in Figure 44. In this example, using the Max Value heuristic on its own would use up all the energy of the UAV before the first 12 hours have passed. After the first surveil using Max Value has ended, the filtering technique detects that the UAV has been consuming energy quickly

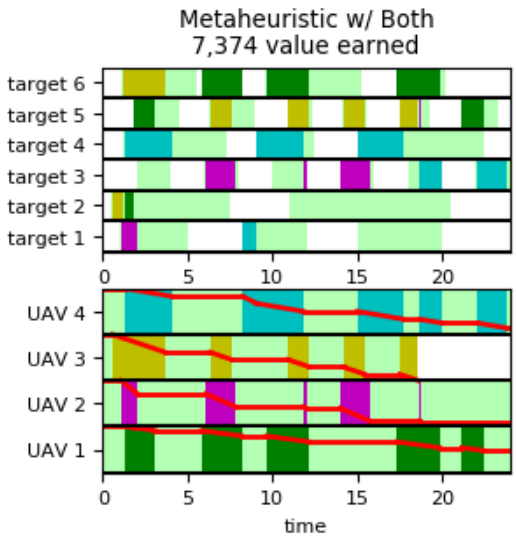


Figure 45. The mapping produced by the Metaheuristic with Preemption and Filtering for our example scenario in Table 9 and Table 10. This mapping earns a total of 7,374 value during the 24 hours we consider. This mapping combines the benefits of Preemption to quickly switch to high priority targets and the benefits of Filtering by conserving energy until later in the day compared to Figure 43. The red lines represent the percentage of remaining energy for each UAV. Colors in this figure have the same meaning as in Figure 43.

and will run out of energy before the end of the day and begins aggressively removing options from the heuristic, resulting in the UAV doing nothing. Near the end of the day, a high priority target arrives and there is still energy remaining to surveil the new target.

The first step of filtering is to calculate the fraction of the day that has passed (δ) and the fraction of the UAV's energy that has been consumed (ε). We define the threshold factor (F) to be the unitless value ε / δ. The base VPE threshold (τ) is equal to the average VPE of all surveils so far for the UAV. Finally, the VPE threshold (T) is equal to $\tau \times F$. At a mapping event, only surveils with VPE > T are considered for each UAV.

With filtering, there will be some mapping events where a UAV will have all of its options removed. This is intentional, as sometimes there is benefit to doing nothing when none of the targets available for a UAV to surveil are efficient options. In those cases, it is usually better to wait for an efficient option to appear and to leave the available targets for other UAVs that may have sensors that are a better fit for the sensor affinities of the targets. The effects of combining

preemption and filtering are demonstrated in Figure 45. When compared to Figure 43, preemption and filtering allow the UAVs to spend most of the day surveilling the most efficient targets. For example, UAV 1 is not significantly constrained by energy and spends almost all of its time surveilling the high priority target 6; and UAV 3 is significant constrained by energy and spends most of the day surveilling target 5, which is efficient in terms of value per unit of energy consumed.

5.4. Simulation Setup

5.4.1. Generation of Baseline Set of Randomized Scenarios

5.4.1.1. Effect of Energy Consumption Rate

Each scenario that we use to evaluate the heuristics is defined by a set of UAV characteristics and a set of target characteristics. To compare and evaluate the heuristics, we consider a wide variety of scenarios to understand the kinds of scenarios for which each heuristic is most effective.

We generate 10,000 baseline scenarios by sampling from probability distributions for the number of UAVs and targets in a scenario in addition to the value for each characteristic of the UAVs and targets. In each case, distributions are selected to attempt to model distributions of parameters that may occur in real-world environments. The details of these distributions are as follows.

5.4.1.2. Generating UAVs

The number of UAVs available during the 24-hour period of a scenario is sampled from a Poisson distribution with the Poisson parameter $\lambda = 9$. The characteristics of each UAV are then generated. The total energy available to the UAV is sampled from a beta distribution with a mean of 0.8 and a standard deviation of 15% of the mean. We use beta distributions for many parameters

in this work because many of our UAV and target characteristics are fixed between a minimum and a maximum value. The energy consumption rate for each sensor is sampled from a beta distribution with a mean of 0.05 and a standard deviation of 50% of the mean. The total energy and energy consumption rates are sampled in this way so that UAVs can be expected to operate for an average of 16 hours.

The number of sensors available to each UAV is generated by using a Rayleigh distribution with a scale parameter of 2. Any values below 1 are increased to 1 and any values above 4 are decreased to 4. The sensor type for each sensor is selected using probabilities of 0.5, 0.2, 0.2, and 0.1 for the VIS, SAR, IR, and LIDAR sensor types, respectively. Because each UAV can only have one sensor of each type, a sensor type that has been selected for a UAV is no longer a candidate for that UAV and the next sensor is chosen among the remaining sensors types after normalizing their probabilities so that the sum is 1.0. The quality of each sensor is found using a beta distribution with a mean of 0.6 and a standard deviation of 40% of the mean. This value is then truncated to an integer and is clamped between 1 and 10, inclusive, such that all UAVs have a sensor quality between 1 (worst) and 10 (best).

5.4.1.3. Generating Targets

The number of targets available to surveil during the 24-hour period is obtained using a Poisson distribution with $\lambda = 14$. Because the number of UAVs was generated with $\lambda = 9$, these scenarios in general will be oversubscribed. The priority of each target is first sampled from a gamma distribution with a mean of 4 and a standard deviation of 60% of the mean. The same method described above for sensor qualities is then used to get integers between 1 (worst) and 10 (best). A gamma distribution was used here because the positive skew results in a slightly larger number of high priority targets instead of the highest priority having the smallest number of occurrences.

Table 11. Dynamic event rates

event type	rate (events per day)
add a UAV	1
remove a UAV	1
remove a sensor from a UAV	0.5
modify sensor qualities of a UAV	0.5
add a target	2
remove a target	2
change the priority of a target	4
change the surveillance time of a target	6
add allowed sensor types to a target	6
remove allowed sensor types from a target	6
modify sensor affinities of a target	2

To obtain the required surveillance time for each target, we use a uniform distribution ranging from 1 to 3 hours.

Differing from what was used for UAVs, we obtain the number of allowed sensor types for each target by adding 1 to the value obtained from a binomial distribution with $p = 0.5$ and $n = 3$. The allowed sensor types selected to match this number are uniformly selected from VIS, SAR, IR, and LIDAR. To get the sensor affinity for each sensor type, we use a beta distribution with a mean of 0.7 and a standard deviation of 30% of the mean and use the same method described above for sensor qualities to get integers between 1 (worst) and 10 (best).

Surveillance intervals are arranged such that the average duration of an interval is three hours and the average time between two intervals is one hour. Starting from time 0, the start time of the first interval is found by sampling an exponential distribution with a mean of one hour and the end of the interval is found by sampling from a gamma distribution with a mean of three hours and a standard deviation of 20% of the mean. This same process is then repeated from the end of the first interval and continues until the next interval would start after the end of the day (24 hours). Note that although these intervals are generated statically in advance, they are dynamic in our system model as described in Subsection 5.2.3 and the heuristics are not aware of where the future intervals will be.

5.4.1.4. Generating Dynamic Events

We utilize Poisson processes to generate the dynamic events for our scenarios. Poisson processes are commonly used to model the occurrences of independent events with a known mean rate. In our baseline set of scenarios, the expected rate of each type of dynamic event is shown in Table 11. For each of the dynamic event types, we model an independent Poisson process where the time between each occurrence of an event of that type is sampled from an exponential distribution with λ equal to the expected rate of events of that type from Table 11.

Each dynamic event that occurs uses methods similar to those described earlier in this section to determine the specific dynamic changes that will occur. If a UAV or target is to be added to the scenario, then a new UAV or target is generated as described in Subsection 5.4.1.2 or 5.4.1.3, respectively. The other event types will affect existing UAVs or targets. For these event types, the UAV or target that is affected is selected randomly (using a uniform distribution).

If a sensor type would be removed from a UAV, that sensor type is selected randomly from the set of sensor types available on the UAV (using a uniform distribution). If the UAV only has one sensor type available, then this is equivalent to removing the UAV from the scenario. When the sensor qualities of a UAV are dynamically changed, they are resampled as described in Subsection 5.4.1.2 as if a new UAV were being created.

The process for changing the characteristics of a target is similar. If the event would affect the sensor affinity or allowed sensor types of a target, these are handled in the same way as sensor quality and sensor types for a UAV, except that new allowed sensor types may be added to a target if it does not already allow all four sensor types that we model. When a new type is added, it is randomly selected from the sensor types that are currently not allowed on the target (using a uniform distribution). When this is done, a sensor affinity for that type is also sampled for the type

as described in Subsection 5.4.1.3. Finally, events that dynamically change either the priority or surveillance time of a target simply resample the quantities as described in in Subsection 5.4.1.3.

5.4.2. Generation of Additional Scenarios for Parameter Sweeps

Because the baseline set of 10,000 scenarios in Subsection 5.4.1 may have characteristics that are favorable to the performance of individual heuristics, we use parameter sweeps to evaluate the heuristics in a diverse set of environments. We generate 20 sets of 10,000 scenarios each for the parameter sweeps of six characteristics of the environment. The characteristics we vary are the mean number of targets in a scenario (two sets in addition to the baseline), the mean number of UAVs in a scenario (two sets in addition to the baseline), the mean rate of energy consumption for sensors (three sets in addition to the baseline), the mean rate at which dynamic events occur (three sets in addition to the baseline), the standard deviation of the priority of targets (four sets in addition to the baseline), and the fixed interval at which mapping events occur (five sets in addition to the baseline). The number of sets for each characteristic were selected such that the impact of each characteristic on the performance of the heuristics is clearly demonstrated.

We examine the effect of varying the number of targets and number of UAVs by generating scenarios for the cases with λ values of 10, 14, and 18 for the number of targets, and 5, 9, and 13 for the number of UAVs. We vary the mean energy consumption of sensors with scenarios where the mean is 0.05, 0.1, 0.15, and 0.2. The rate of dynamic events is varied by multiplying the rates given in Table 11 by 0, 1, 4, and 16. The coefficient of variation of target priority is varied between 0.2, 0.4, 0.6, 0.8, and 1.0. Finally, to analyze the effect of the mapping interval, we consider mapping intervals of 1, 5, 10, 30, 60, and 120 minutes.

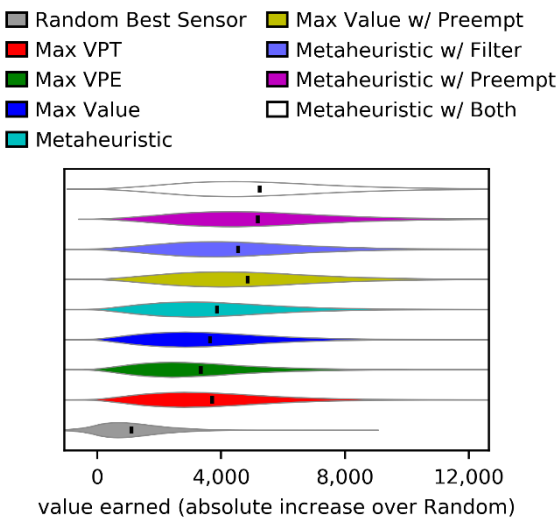


Figure 46. A violin plot showing the difference between the surveillance value earned by each heuristic and the Random heuristic for the set of 200,000 small scenarios described in Subsections 5.4.1 and 5.4.2. The mean difference for each heuristic is indicated by the black marker in each distribution.

5.4.3. Generation of Large-Scale Scenarios

We also generate a baseline set of 500 scenarios in the same way as the 10,000 described in Subsection 5.4.1, except that the scenario is ten times as large on average. Specifically, all characteristics of the scenario are generated as described in Subsection 5.4.1, except that there is an average of 90 UAVs and 140 targets. In addition, the dynamic events are generated as described in Subsection 5.4.1.3, but with expected rates of events that are ten times as high as listed in Table 11. This baseline set of scenarios is then expanded following the same process described in Subsection 5.4.2 with the number of targets and UAVs scaled up by ten times. This set of scenarios is used to explore how effectively the heuristics scale when large scenarios are considered.

5.5. Simulation Results

5.5.1. Results for Randomized Set of Small Scenarios

5.5.1.1. Overview

As described in Subsection 5.4.2, the results shown in this section consist of parameter sweeps where the means of the distributions described in Subsection 5.4.1 are varied. Figure 46 is a violin plot, which shows the overall performance of each heuristic in all 200,000 of the scenarios we generated in Subsections 5.4.1 and 5.4.2. This overview of our results indicates that the Metaheuristic is among the best value-based heuristics without preemption or filtering modifications. Additionally, when modified with preemption or filtering, the average performance of the Metaheuristic improves significantly.

5.5.1.2. Effect of Energy Consumption Rate

In Figure 47, the subset of results where we vary the rate of energy consumption is shown. When the rate of energy consumption is low, the preemption modification results in heuristics that perform significantly better than the others. This is because in scenarios where UAVs can surveil targets for the entire day without running out of energy, it is most important to ensure that surveils on high priority targets begin as soon as possible with the UAVs that have high quality sensors with the best affinity for those targets. With preemption, heuristics can immediately start surveilling those targets with the optimal UAVs with no delay. When the rate of energy consumption increases, preemption still brings significant benefits, but the filtering modification becomes more effective than preemption because it is also important to ensure that the energy of the UAVs that are best for high priority targets later in the day is conserved.

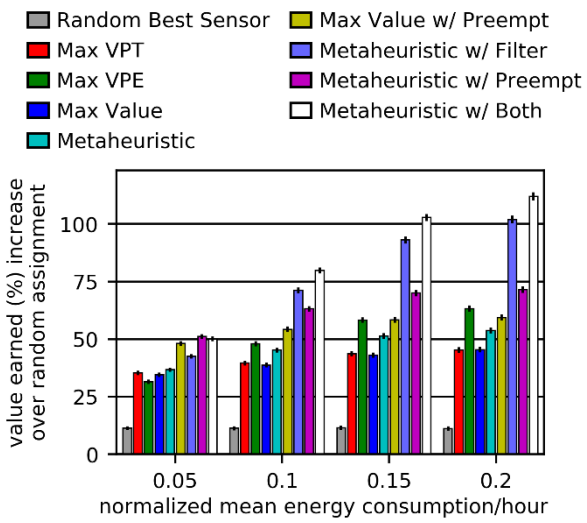


Figure 47. A comparison of the percentage increase in surveillance value earned when compared to the Random heuristic in 10,000 small randomized scenarios. The mean rate of energy consumption per hour is varied from the baseline set of scenarios with a rate of 0.05 normalized units of energy per hour. Except for the rate of energy consumption, the other characteristics of the scenario use the values from the baseline case described in Subsection 5.4.1. The 95% mean confidence intervals are shown for each bar.

When the rate of energy consumption is low (e.g., in the case with a mean energy consumption rate of 0.05 units of energy per hour), the filtering technique is less effective than preemption. A reason for this is because filtering causes the UAVs to sometimes not surveil any target to conserve energy and when the rate of energy consumption is very low, this can be counterproductive.

These results appear very different from Figure 46 because Figure 46 displays the distribution of results from all 200,000 of the small scenarios, while the scenarios shown in Figure 47 are only 40,000 of those scenarios. The remaining 160,000 scenarios use a mean rate of energy consumption of 0.05 units of energy per hour, which matches the first set of bars in Figure 47. When comparing this first set of bars to the overall results in Figure 46, the pairwise relative performance of all heuristics appears similar.

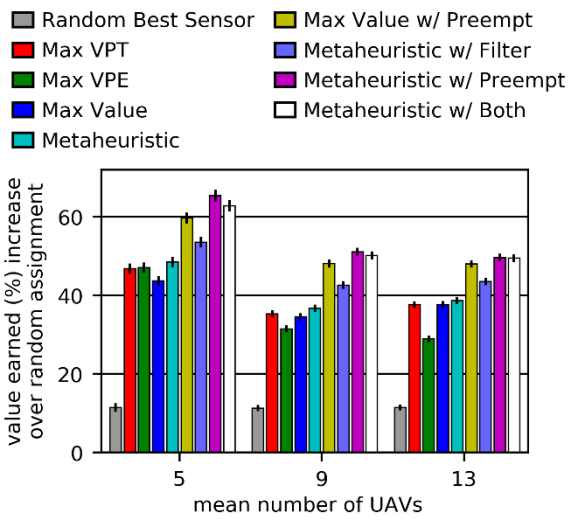


Figure 48. A comparison of the percentage increase in surveillance value earned when compared to the Random heuristic in 10,000 small randomized scenarios. The mean number of UAVs is varied from the baseline set of scenarios with a mean of 9 UAVs. Except for the number of UAVs, the other characteristics of the scenario use the values from the baseline case described in Subsection 5.4.1. The 95% mean confidence intervals are shown for each bar.

5.5.1.3. Effect of Number of UAVs and Targets

When the number of UAVs is varied as shown in Figure 48, our results show that the Metaheuristic with preemption performs among the best heuristics in all cases. When there are many UAVs, energy is not a significant constraint because there will be unused UAVs still available when some UAVs start running out of energy, which results in similar relative performance of heuristics to the cases with a low rate of energy consumption shown in Figure 47.

5.5.1.4. Effect of Coefficient of Variation of Target Priorities

As the coefficient of variation of target priorities increases for the results in Figure 49, there are three notable characteristics of the surveillance value earned by the heuristics. First, the Random and Random Best Sensor heuristics perform equally for all five values of the coefficient of variation that we considered (0.2, 0.4, 0.6, 0.8, and 1.0). This is expected because these heuristics select targets without considering priority. All value-based heuristics perform better as

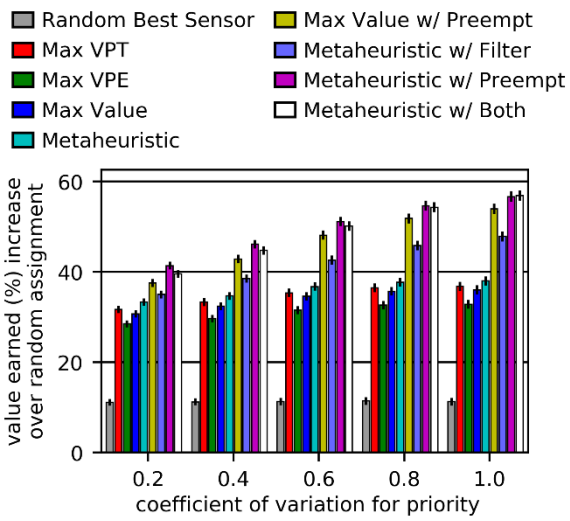


Figure 49. A comparison of the percentage increase in surveillance value earned when compared to the Random heuristic in 10,000 small randomized scenarios. The coefficient of variation is varied from the baseline set of scenarios with coefficient of variation of 0.6. Except for this coefficient of variation, the other characteristics of the scenario use the values from the baseline case described in Subsection 5.4.1. The 95% mean confidence intervals are shown for each bar.

the coefficient of variation increases with the preemptive heuristics performing the best overall in all cases. The preemption and filtering modifications improve the most in performance when the variance in priority increases because they help to ensure that all high priority targets are surveilled. However, the Metaheuristic with preemption is still among the best overall heuristics because it is more likely that high priority targets will become available while low priority targets are being surveilled in the scenarios with a higher coefficient of variation.

5.5.2. Results for Randomized Set of Large Scenarios

The results in this subsection were generated using the set of large scenarios detailed in Subsection 5.4.3. As shown in Figure 50, the most significant differences between the overall results when compared to the smaller scenarios shown in Figure 46 are that: (a) the Metaheuristic with both preemption and filtering is the best overall heuristic, (b) the filtering modification performs equal to preemption on its own when applied to the Metaheuristic, and (c) the performance of all value-based heuristics relative to Random has increased. Preemption performs

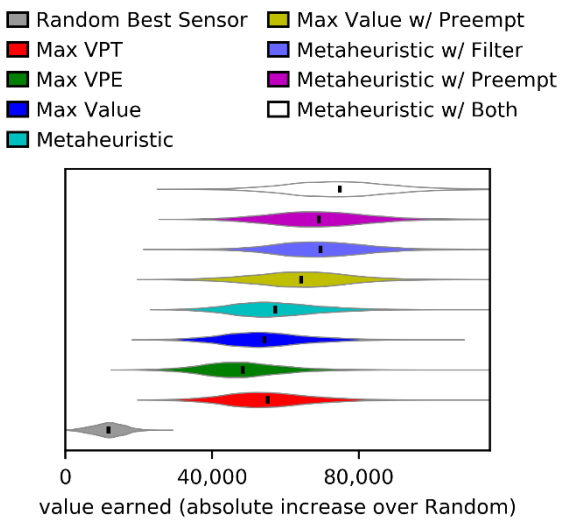


Figure 50. A violin plot showing the difference between the surveillance value earned by each heuristic and the Random heuristic for the set of 10,000 large scenarios described in Subsection 5.4.3. The mean difference for each heuristic is indicated by the black marker in each distribution.

well for the same reasons given in Section 5.5.1. The filtering modification performs significantly better in the larger scenarios because there is a larger number of UAVs and targets. This increases the chance that there are targets that have characteristics that are highly efficient for each UAV, which means it is more important to save energy to surveil those targets.

5.5.3. Discussion of Results

The results in Subsections 5.5.1 and 5.5.2 indicate that depending on the scenario, either Max Value or Max Value Per Energy is an effective real-time heuristic for maximizing surveillance value. The Metaheuristic combines the strengths of both heuristics and is effective in all scenarios. When considering scenarios where the energy of UAVs will not be fully consumed during the day, Max Value Per Energy is ineffective. Based on these results, our proposed Metaheuristic is the best option to use in all cases where the characteristics of the scenario may change unexpectedly.

Additionally, the results demonstrate that both of our proposed modifications for the heuristics, preemption and filtering, improve performance of the Metaheuristic on average. The preemption

modification always results in significant improvement in average surveillance value earned because it allows immediate response when options that would result in higher value surveils become available. For example, UAVs can immediately begin surveilling high priority targets without completing their currently active surveil first. The filtering modification performs extremely well in scenarios where a significant portion of the UAVs have options in the scenario that are significantly more efficient than others in terms of value earned per unit of energy consumed. This is because filtering conserves some of the energy for each UAV until later in the day when these efficient options that result in high surveillance value may be available.

In our simulations, mapping events for the slowest heuristic, the Metaheuristic with both preemption and filtering, took an average of less than 10 milliseconds for each mapping to complete for the small set of scenarios and an average of less than 1 second each for the large set of scenarios. This demonstrates that any of these heuristics can be used to find mission schedules in real-time.

5.6. Related Work

Developing a complete mission schedule for UAVs involves solving multiple problems, many of which have been studied in the past such as planning the specific routes used by UAVs, which we do not consider in this study, and assigning specific tasks to UAVs. Some studies solve these problems through time-consuming optimization techniques such as mixed integer linear programming (MILP), while others use techniques ranging from time-intensive metaheuristics like genetic algorithms (GAs) to fast and efficient greedy heuristics to find effective solutions.

In [5], mission planning is divided into two subproblems: task scheduling and route planning. The task scheduling problem is the one we consider in our study. The task scheduling problem is solved using an MILP approach to minimize the completion time of all tasks as opposed to our

work that aims to maximize surveillance value. Similarly, in [6] a swarm of UAVs is also optimized to perform tasks while minimizing total completion time using an MILP approach. In [7], UAVs are assigned to attack and attempt to destroy clusters of targets through expressing three objective measures into one weighted measure (the success probability of the attack, the cost of the attack as a function of fuel consumption and risk to the UAV, and how well the timing of the attack will match a desired window), which is used to apply integer programming methods to find a solution. No-fly zones are considered in [8], which compares MILP and heuristic techniques to solve a task assignment problem where UAVs must complete a sequence of tasks. A solution here is represented by a directed acyclic graph (DAG). In [9], possible solutions for mapping a UAV to any combination of targets is represented by a decision tree, where moving from the root to a node represents assigning the UAV to the target corresponding to that node. A best first search (BFS) method is used to find solutions to this problem. In [10], a fleet of UAVs must be used to provide continuous 5G network coverage to the region of interest. The goal of this work is to determine both the required number of UAVs to guarantee coverage and to create a mission schedule for these UAVs. This is accomplished through a brute-force combinatorial technique to find the optimal solution, which is applicable in this case due to the size of problems considered. The most significant difference between these studies and our work is that we consider scenarios where decisions must be made in real-time throughout the day in a dynamic environment based on a mathematical model of a performance metric.

Mission planning for UAVs is sometimes studied as an orienteering problem [11], [12]. For example, in [11] the authors utilize a model where UAVs originate from a depot and gain profit from traveling a path through nodes and back to the depot. Robust optimization techniques are used to maximize profit while taking uncertainty into account to avoid running out of fuel early.

This work differs significantly from ours because distance between targets and UAVs is considered and the focus is on optimization of UAV movement instead of sensing. In [12], UAVs again depart from a depot, but before departure they can select a specific set of sensors, which will impact their weight and the information they can gather. The authors solve this problem using both an MILP approach when ample time is available for finding solutions and several heuristic techniques for larger problem sizes that cannot be solved in a reasonable amount of time using the MILP approach. Our work differs significantly from these studies in part because a full mission plan is generated by the MILP approach and it is not modified during the day. In our work, the heuristics dynamically schedule UAVs to assign targets many times throughout the day and these decisions depend on the current state of the scenario. Additionally, our work considers the energy consumption of each sensor, which can greatly impact scheduling decisions.

Some studies have a greater focus on motion planning of the UAVs, which is outside the scope of our contribution in this work [13]–[20]. Our focus is on characterizing UAVs, sensors, and targets to develop a mathematical model that can be used as a system-wide performance measure that quantifies the success of surveils. We use this model as a basis for designing, evaluating, and comparing various dynamic mission scheduling heuristics through extensive simulation studies.

5.7. Conclusions and Future Work

We created a novel metric to quantity system-wide performance of UAVs surveilling targets in dynamic scenarios. To effectively compare mission schedules using simulations, we also detailed a model for generating randomized scenarios where a heterogeneous set of UAVs surveils a set of heterogenous targets.

We designed a set of value-based heuristics used to conduct mission planning for UAV surveillance of a set of targets (Max Value, Max Value Per Time, Max Value Per Energy, and the

Metaheuristic). We conducted a simulation study to evaluate, analyze, and compare these heuristics in a variety of scenarios. We found that while Max Value and Max Value Per Energy are each good heuristics for a subset of the scenarios considered, the Metaheuristic found solutions with among the highest surveillance value for all scenarios.

In addition, we developed two modifications to these heuristics to improve their performance (preemption and filtering). Our simulations demonstrate that both preemption and filtering can significantly improve the performance of our heuristics and can be combined to take advantage of the benefits provided by both modifications. We found that preemption is effective in all environments we considered and performs better than filtering on average; however, in environments where there is little energy available to the UAVs, the filtering modification performs much better than the preemption modification. These results make it clear that it is an effective choice to always employ the preemption modification and that filtering should also be included in energy-constrained environments.

References

[1] J. Crowder and J. Carbone, "Autonomous Mission Planner and Supervisor (AMPS) for UAVs," *20th International Conference on Artificial Intelligence (ICAI '18)*, pp. 195–201, July 2018.

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

[3] D. Machovec, B. Khemka, N. Kumbhare, S. Pasricha, A. A. Maciejewski, H. J. Siegel, A. Akoglu, G. A. Koenig, S. Hariri, C. Tunc, M. Wright, M. Hilton, R. Rambharos, C. Blandin, F. Fargo, A. Louri, and N. Imam, "Utility-Based Resource Management in an Oversubscribed Energy-Constrained Heterogeneous Environment Executing Parallel Applications," *Parallel Computing*, vol. 83, pp. 48–72, 2019.

[4] B. Khemka, R. Friese, S. Pasricha, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, S. Powers, M. Hilton, R. Rambharos, and S. Poole, "Utility Maximizing Dynamic Resource Management in an Oversubscribed Energy-Constrained Heterogeneous Computing System," *Sustainable Computing: Informatics and Systems*, vol. 5, pp. 14–30, 2015.

[5] J. J. Wang, Y. F. Zhang, L. Geng, J. Y. H. Fuh, and S. H. Teo, "Mission Planning for Heterogeneous Tasks with Heterogeneous UAVs," *13th International Conference on Control Automation Robotics & Vision (ICARCV)*, pp. 1484–1489, Mar. 2014.

[6] C. Schumacher, P. Chandler, M. Pachter, and L. Pachter, "UAV Task Assignment with Timing Constraints," *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 9 pp., Jun. 2003.

[7] J. Zeng, X. Yang, L. Yang, and G. Shen, "Modeling for UAV Resource Scheduling Under Mission Synchronization," *Journal of Systems Engineering and Electronics*, vol. 21, no. 5, pp. 821–826, 2010.

[8] S. Leary, M. Deittert, and J. Bookless, "Constrained UAV Mission Planning: A Comparison of Approaches," *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pp. 2002–2009, Nov. 2011.

[9] M. Faied, A. Mostafa, and A. Girard, "Vehicle Routing Problem Instances: Application to Multi-UAV Mission Planning," *AIAA Guidance, Navigation, and Control Conference*, 12 pp., Aug. 2010.

[10] C. Tipantuña, X. Hesselbach, V. Sánchez-Aguero, F. Valera, I. Vidal, and B. Nogales, "An NFV-Based Energy Scheduling Algorithm for a 5G Enabled Fleet of Programmable Unmanned Aerial Vehicles," *Wireless Communications and Mobile Computing*, 20 pp., 2019.

[11] L. Evers, T. Dollevoet, A. I. Barros, and H. Monsuur, "Robust UAV Mission Planning," *Annals of Operations Research*, vol. 222, pp. 293–315, 2012.

[12] F. Mufalli, R. Batta, and R. Nagi, "Simultaneous Sensor Selection and Routing of Unmanned Aerial Vehicles for Complex Mission Plans," *Computers & Operations Research*. vol. 39, pp. 2787–2799, 2012.

[13] W. Chung, V. Crespi, G. Cybenko, and A. Jordan, "Distributed Sensing and UAV Scheduling for Surveillance and Tracking of Unidentifiable Targets," *Proceedings of SPIE 5778, Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense IV*, pp. 226–235, May 2005.

[14] D. Pascarella, S. Venticinque, and R. Aversa, "Agent-Based Design for UAV Mission Planning," 8th International Conference on P2P, *Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pp. 76–83, Oct. 2013.
[15] J. Kim, B. D. Song, and J. R. Morrison, "On the Scheduling of Systems of UAVs and Fuel Service Stations for Long-Term Mission Fulfillment," *Journal of Intelligent & Robotic Systems*, vol. 70, pp. 347–359, 2013.
[16] C. R. Atencia, J. D. Ser, and D. Camacho, "Weighted Strategies to Guide a Multi-Objective Evolutionary Algorithm for Multi-UAV Mission Planning," *Swarm and Evolutionary Computation*, vol. 44, pp. 480–495, 2019.
[17] Z. Zhen, Y. Chen, L. Wen, and B. Han, "An Intelligent Cooperative Mission Planning Scheme of UAV Swarm in Uncertain Dynamic Environment," *Aerospace Science and Technology*, vol. 100, 16 pp., 2020.
[18] G. Q. Li, X. G. Zhou, J. Yin, and Q. Y. Xiao, "An UAV Scheduling and Planning Method for Post-Disaster Survey," ISPRS - *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XL-2, pp. 169–172, 2014.
[19] W. Yang, L. Lei, and J. Deng, "Optimization and Improvement for Multi-UAV Cooperative Reconnaissance Mission Planning Problem," *11th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*, pp. 10–15, Dec. 2014.
[20] A. Thibbotuwawa, G. Bocewicz, G. Radzki, P. Nielsen, and Z. Banaszak, "UAV Mission Planning Resistant to Weather Uncertainty," *Sensors*, vol. 20, no. 2, 24 pp., 2020.
[21] R. D. Friese, J. A. Crowder, H. J. Siegel, and J. N. Carbone, "Bi-Objective Study for the Assignment of Unmanned Aerial Vehicles to Targets." *20th International Conference on Artificial Intelligence (ICAI '18)*, pp. 207–213, July 2018.
[22] R. D. Friese, J. A. Crowder, H. J. Siegel, and J. N. Carbone, "Surveillance Mission Planning: Model, Performance Measure, Bi-Objective Analysis, Partial Surveils," *21st International Conference on Artificial Intelligence (ICAI '19)*, 7 pp., July 2019.
[23] D. Machovec, J. A. Crowder, H. J. Siegel, S. Pasricha, and A. A. Maciejewski, "Dynamic Heuristics for Surveillance Mission Scheduling with Unmanned Aerial Vehicles in Heterogeneous Environments," *22nd International Conference on Artificial Intelligence (ICAI '20)*, 21 pp., July 2020.
[24] D. Machovec, H. J Siegel, J. A. Crowder, S. Pasricha, A. A. Maciejewski, and R. D. Friese, "Surveillance Mission Scheduling with Unmanned Aerial Vehicles in Dynamic Heterogeneous Environments," journal submission

Chapter 6

Conclusion

This research explored resource management for HPC systems and surveillance mission planning problems. The contributions can be divided into three main categories. First, system-wide metrics and system models were designed and improved for each environment. Without meaningful metrics and models, meaningful analysis of techniques to solve the problems is not possible. Accurately modeling each environment is important because characteristics of the environment that may otherwise be overlooked can have a significant impact. Finally, simulators were developed for each environment to allow analysis, comparison, and evaluation of novel and existing techniques used to solve each problem.

A Value of Service (VoS) metric was developed, which measures total value earned as a function of energy consumption for each task in addition to when a task is completed. This metric allows fine-tuning of the impact of performance versus energy when calculating the VoS for each category of a tasks and the time of day when tasks is submitted. Tasks were allocated to virtual machine (VM) configurations, which specified the number of cores and amount of memory that will be assigned to a task type. Resource management heuristics were designed that attempt to maximize the system-wide VoS metric. These heuristics were evaluated and compared in a variety of environments using simulations, which were then validated with focused experiments on an IBM HS22 blade server. The simulations and experiments both showed that Max Value-per-Total

Resource (Max VPTR) outperforms the Simple with dropping and placeholder heuristic and its variations.

Resource management for high performance computing (HPC) environments was explored in an energy-constrained environment where tasks are parallel and execute on multiple cores at once. Utility-aware resource allocation heuristics (Max Util, Max UPT, Max UPR, Max UPE, and two metaheuristics) were designed and evaluated along with dropping and filtering techniques that were applied to the heuristics. Performance was measured in terms of the total system utility earned from the completion of parallel tasks in an oversubscribed HPC environment with an energy constraint. The novel concept of place-holders was presented and the new energy-per-resource filtering technique allowed the utility-based heuristics to achieve significantly higher system utility than popular scheduling techniques from literature that do not consider utility functions and heterogeneity. The Max UPR with place-holders heuristic and energy filtering earned utility comparable to the energy-aware Max UPE with place-holders heuristic, demonstrating that energy filtering can improve the energy-efficiency of heuristics designed to greedily maximize utility earned. In addition, both metaheuristics, the Event-Based Metaheuristic and the Task-Based Metaheuristic, earned the highest utility in all environments where there was a steady rate of task arrivals. When the task arrival pattern has significant variance, the Max UPE heuristic performed best in the most energy-constrained environments that were considered.

Preemption-capable resource management heuristics were considered for HPC environments with urgent serial tasks of high importance. To model these environments, realistic patterns where tasks arrive in bursts were considered and utility functions were modified to capture the urgency of the important tasks. To meet the needs of these tasks, six preemption-capable heuristics were designed and were shown to significantly outperform Random and the common FCFS scheduling

technique. In addition, the preemption-capable heuristics were able to outperform the utility-aware heuristics without preemption (Max Util and Max UPT).

A UAV surveillance mission scheduling problem was studied, in which a novel metric that quantifies the system-wide performance of UAVs surveilling targets in dynamic scenarios was created. Multiple factors are considered for each surveil to obtain a value for that surveil, including the quality of information gathered (e.g., image resolution), the importance of the surveilled target, and the relevance of the information obtained for that target. The values for each surveil are combined into the single system-wide performance measure called surveillance value. A novel Metaheuristic was developed, which can be enhanced with a combination of preemption and filtering techniques. A probabilistic model for generating randomized scenarios was designed, which enabled analysis, comparison, and evaluation of mission scheduling techniques through an extensive simulation study. This study demonstrated that the Metaheuristic with preemption and filtering found schedules with among the highest surveillance value for all scenarios.

Chapter 7

Suggested Future Work

There are many possible directions for extensions of the research described in the previous chapters. One possible extension is the applications of popular optimization techniques such as reinforcement learning could be applied to resource management for any of the environments described in this report.

The metaheuristics and filtering techniques in Chapters 3 and 5 are limited because they assume that the optimal rate of energy consumption would be a linear rate of consumption throughout the day. These techniques could be extended to support other patterns where the rate of energy consumption varies throughout the day. For example, many HPC systems will usually have specific times where more newly submitted jobs are expected, which would significantly increase the potential rate of energy consumption at those times.

Chapter 4 has room for improvement by extending the preemption-capable heuristics and models to support parallel jobs, which significantly increases the computational complexity of finding which combinations of assignments and preemptions should occur, especially when considering reservations and place-holders that extend far into the future. Additionally, it is important to consider the amount of overhead required due to suspending and resuming jobs when studying preemption-capable resource management techniques.

The work in Chapter 5, in which dynamic heuristics are used to maximize the surveillance value gained by UAVs surveilling targets, could be enhanced by considering the relative positions

of all UAVs and targets. This includes dynamically moving the UAVs to the targets that will be surveilled and modeling the energy consumption of the UAVs due to flight in addition to the energy consumption of the UAVs' sensors. If targets move throughout the day, this will require adjustments to the mission schedule to account for this. This could also include modeling depots where the UAVs start and finish each day, allowing for UAVs to return to the depot early to refuel. This will require new characteristics for UAVs, such as their current position, the velocity at which they can travel between targets and the rate of energy consumption consumed by flight, and how much time it takes to refuel the UAV. Additionally, there would be new characteristics for targets, such as their position, how quickly the targets can move, and whether there are constraints on when they can move.