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

Dylan Machovec[1], Sudeep Pasricha[1,2], Anthony A. Maciejewski[1], Howard Jay Siegel[1,2], Gregory A. Koenig, Michael Wright[4], Marcia Hilton[4], Rajendra Rambharos[4], Thomas Naughton[3], and Neena Imam[3]

[1]Department of Electrical and Computer Engineering
[2]Department of Computer Science
Colorado State University
Fort Collins, CO 80523, USA

[3]Oak Ridge National Laboratory
Oak Ridge, TN 37831, USA
[4]Department of Defense
Washington, DC 20001, USA

djmachov@rams.colostate.edu, {sudeep,aam,hj}@colostate.edu, koenig@acm.org, michael.wright4@comcast.net, mmskizig@verizon.net, jendra.rambharos@gmail.com, naughtont@ornl.gov, imamn@ornl.gov

***Abstract*—We design resource management heuristics that assign serial tasks to the nodes of a heterogeneous high performance computing (HPC) system. The value of completing these tasks is modeled using monotonically decreasing utility functions that represent the time-varying importance of the task. The value of completing a task is equal to its utility function at the time of its completion. The overall performance of this system is measured using the total utility earned by all tasks during some interval of time. To maximize the performance of such a system where the preemption of tasks is possible, we have designed, analyzed, and compared a set of resource allocation heuristic techniques. We combine two utility-aware heuristics with three different preemption techniques to create six preemption-capable heuristics. We also consider the two utility-aware heuristics without preemption. We use simulation studies to evaluate this set of eight heuristics and compare them with an FCFS heuristic, which is often used in real systems, and random assignments. In general, our set of eight heuristics is able to significantly outperform the comparison heuristics, and the preemption-capable heuristics are able to significantly increase the utility earned compared to the heuristics that do not use preemption. We analyze the performance tradeoffs among the different preemption-capable heuristics under a variety of oversubscribed workload environments.**

Keywords—heterogeneous computing; utility functions; resource management heuristics; scheduling; preemption

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-public-accessplan). 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.

I. Introduction

We study resource management for <u>h</u>igh <u>p</u>erformance <u>c</u>omputing (<u>HPC</u>) environments. The workload of such a system can be <u>dynamic</u>, i.e., the arrival pattern of tasks in the system is not known in advance. In addition, HPC systems are often <u>oversubscribed</u>, 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 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 <u>critical</u> <u>tasks</u> and all other tasks to be <u>non</u>-<u>critical</u> <u>tasks</u>. 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 II, 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 III. Section IV describes the procedure that was used to generate the specific systems and workloads that were simulated and Section V presents the results of those simulations. Related work is discussed in Section VI. Finally, we conclude and discuss ideas for future work in Section VII.

II. Environment Description

A. *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.

B. *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 1. 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.

C. 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 1. 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].

TABLE I. 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

D. 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

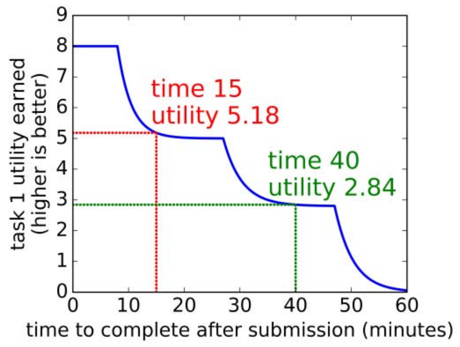

Figure 1. An example of a utility function for task 1. If task 1 were to finish its execution at time 15, it would earn 5.18 utility. If its execution completes at time 40, it will earn 2.84 utility. This figure was taken from our previous work [21].

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 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 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.

E. 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 III.

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.

III. Resource Management

A. 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 III.B) and then one of the heuristics (described in Subsections III.C, III.D, and III.E) is used to create a mapping of the tasks to the cores. Finally, this mapping is used to preempt and assign tasks.

B. 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.

C. Comparison Heuristics

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.

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.

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.

D. Utility-Aware Heuristics

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 III.E.

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].

3) Maximum Util: Max Util, shown in Algorithm 1, operates in two phases. In the first phase (lines 2-4 of Algorithm 1), 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 1) 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.

Algorithm 1. 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
```

Algorithm 2. 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
```

E. Preemption Techniques

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 III.D 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.

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 2. Specifically, during the first phase of the heuristic (lines 2-12 of Algorithm 2) 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-

Algorithm 3. 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
```

18 of Algorithm 2) 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.

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 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) 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 3. 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 3), 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 3. 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 3. During the second phase of this heuristic (shown on lines 16-22 of Algorithm 3), 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.

F. 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.

IV. Simulation Setup

A. 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 V. 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 IV.

B. 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.

C. 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 V, 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 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 1 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 2.

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 <u>burst size</u> 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 V assume that all tasks are preemptible and all tasks can be preempted. We will also show one set of results in Section V where we experimented with these preemption flags.

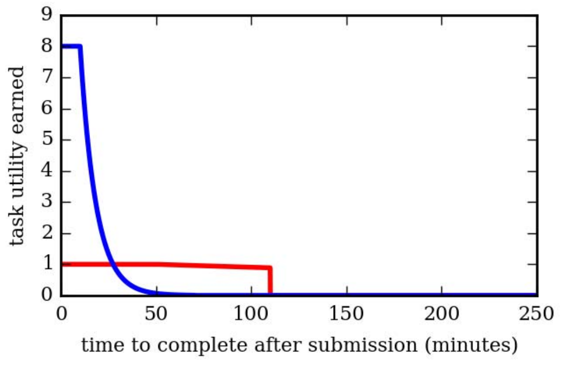

Figure 2. Two utility functions generated using the decaying utility class described in Section IV.C. The blue function is generated for critical tasks and the red function is generated for non-critical tasks.

V. 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 IV. 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 <u>maximum system utility</u> 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 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 3, 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 IV.C. 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 4, 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 4 results, the 20 utility classes described in Subsection VI.C 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%.

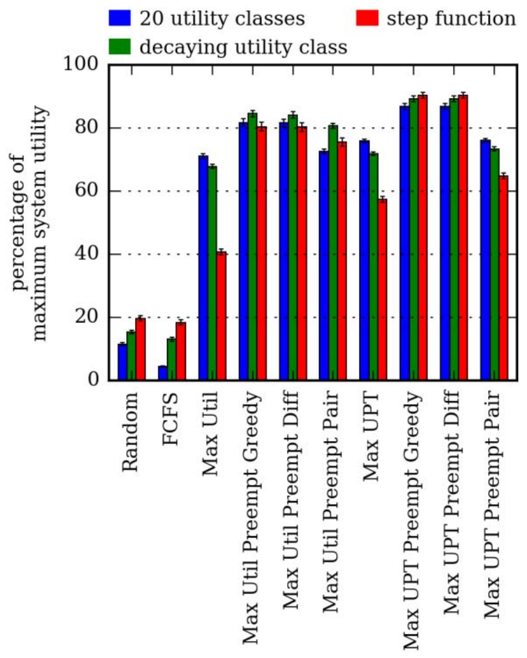

Figure 3. The percentage of maximum system utility is shown for workloads where the utility class used to determine utility functions is varied from step functions, a single decaying utility class, and 20 utility classes as described in Subsection IV.C. The results are shown with 95% mean confidence intervals.

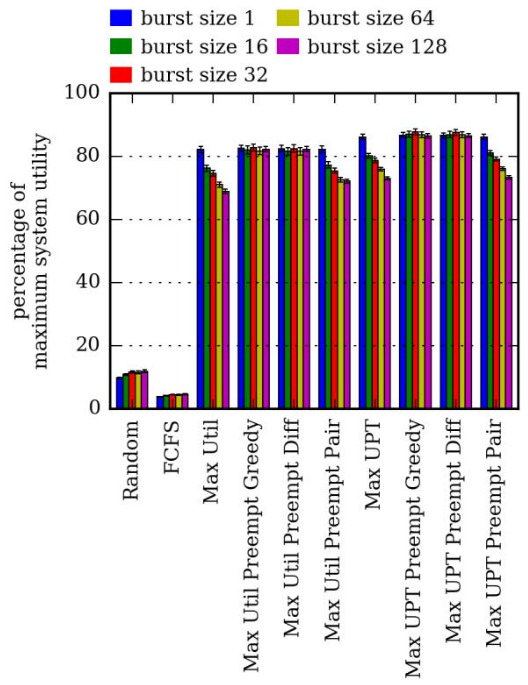

Figure 4. The percentage of maximum system utility earned by each of the heuristics for five different workloads where the burst sizes of tasks arriving are 1, 16, 32, 64, and 128. The results are shown with 95% mean confidence intervals.

In Figure 5, 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 5 results, the 20 utility classes described in Subsection VI.C 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 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 6, 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 6 results, the 20 utility classes described in Subsection VI.C 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 7, the percentage of tasks that are completed by each of the heuristics is shown for the same workloads in Figure 6 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 Figures 3, 4, 5, and 6 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.

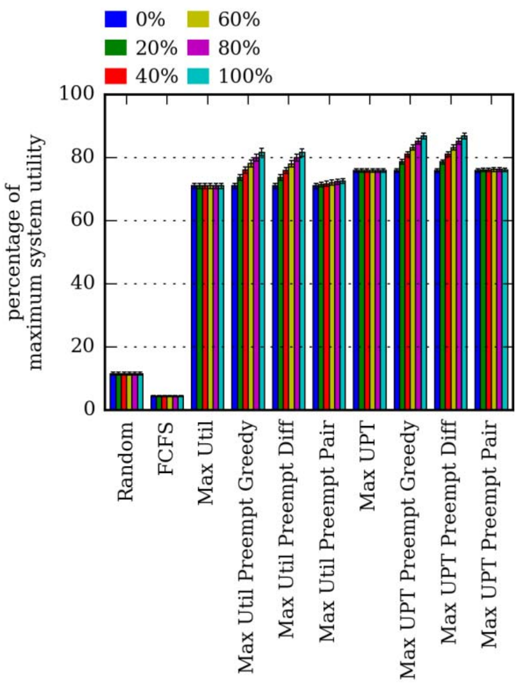

Figure 5. The percentage of maximum system utility earned by each of the heuristics for six different workloads where the percentage of tasks that can preempt and the percentage of tasks that can be preempted are varied over 0%, 20%, 40%, 60%, 80%, and 100%. The results are shown with 95% mean confidence intervals.

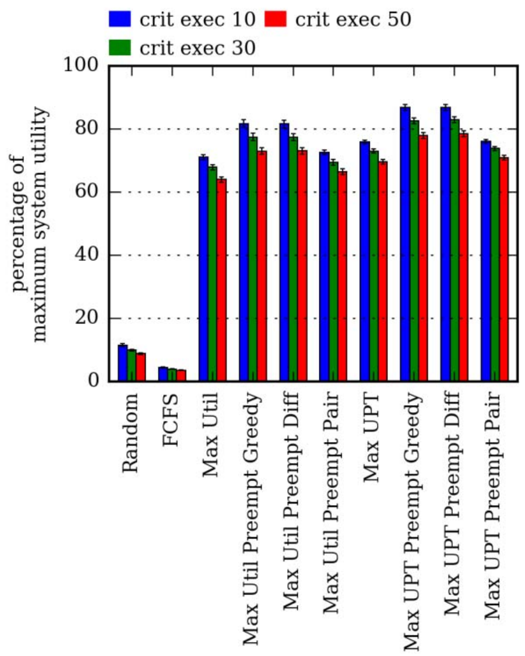

Figure 6. The percentage of maximum system utility earned by each of the heuristics for three different workloads where the average execution times of critical tasks are 10, 30, and 50 minutes. Recall that the execution time of non-critical tasks is 50 minutes. The results are shown with 95% mean confidence intervals.

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 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.

VI. 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 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.

VII. 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.

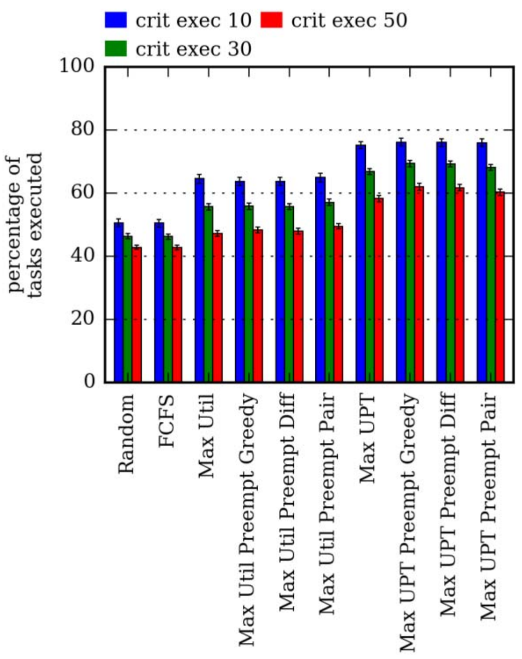

Figure 7. The percentage of tasks completed by each of the heuristics for three different workloads where the average execution times of critical tasks are 10, 30, and 50 minutes. Recall that the execution time of non-critical tasks is 50 minutes. The results are shown with 95% mean confidence intervals.

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.

Acknowledgements

The authors thank Daniel Dauwe and Ninad Hogade for their comments on this work.

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," in *Advances in Computers* vol. 63: *Parallel, Distributed, and Pervasive Computing*, 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, Nov. 2000. Invited.

[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," in *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," in *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, June 2001.

[6] L. Cheng, Q. Zhang, and R. Boutaba, "Mitigating the negative impact of preemption on heterogeneous MapReduce workloads," in *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," in *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," in *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," in *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," in *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," in *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. Klusacek and H. Rudova, "Performance and fairness for users in parallel job scheduling," in *Job Scheduling Strategies for Parallel Processing*, vol. 7698 of Lecture Notes in Computer Science, pp. 235-252. 2012.

[18] C. Leangsuksun, J. Potter, and S. Scott, "Dynamic task mapping algorithms for a distributed heterogeneous computing environment," in *4th IEEE Heterogeneous Computing Workshop (HCW 1995)*, pp. 30-34, Apr. 1995.

[19] D. A. Lifka. "The ANL/IBM SP scheduling systems," in *Job Scheduling Strategies for Parallel Processing*, vol. 949 of Lecture Notes in Computer Science, 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," in *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," in *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.