Utility Functions and Resource Management in an Oversubscribed Heterogeneous Computing Environment

Bhavesh Khemka, Ryan Friese, Luis D. Briceño, Howard Jay Siegel, *Fellow, IEEE*, Anthony A. Maciejewski, *Fellow, IEEE*, Gregory A. Koenig, Chris Groer, Gene Okonski, Marcia M. Hilton, Rajendra Rambharos, and Steve Poole

Abstract—We model an oversubscribed heterogeneous computing system where tasks arrive dynamically and a scheduler maps the tasks to machines for execution. The environment and workloads are based on those being investigated by the Extreme Scale Systems Center at Oak Ridge National Laboratory. Utility functions that are designed based on specifications from the system owner and users are used to create a metric for the performance of resource allocation heuristics. Each task has a time-varying utility (importance) that the enterprise will earn based on when the task successfully completes execution. We design multiple heuristics, which include a technique to drop low utility-earning tasks, to maximize the total utility that can be earned by completing tasks. The heuristics are evaluated using simulation experiments with two levels of oversubscription. The results show the benefit of having fast heuristics that account for the importance of a task and the heterogeneity of the environment when making allocation decisions in an oversubscribed environment. The ability to drop low utility-earning tasks allow the heuristics to tolerate the high oversubscription as well as earn significant utility.

Index Terms—Utility function, resource management heuristics, heterogeneous computing

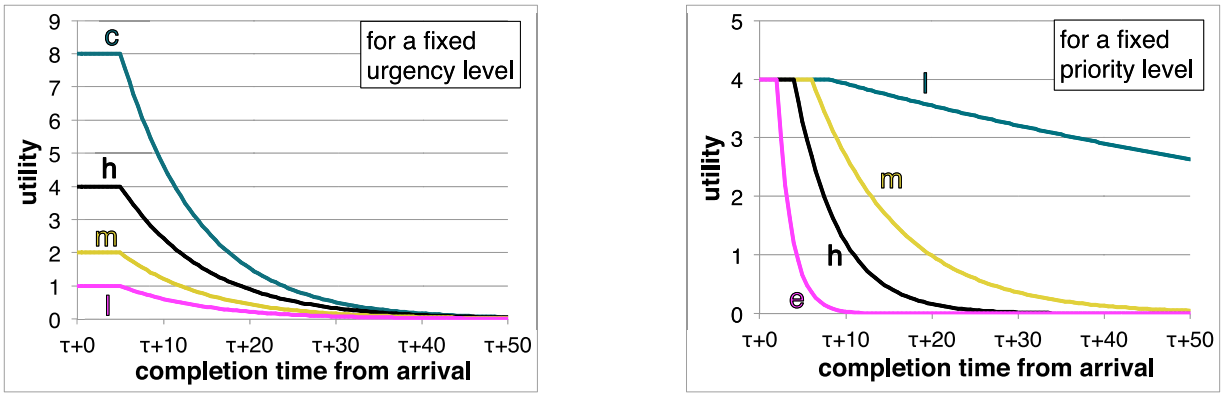
1 Introduction

A utility function for a task describes the value of completing the execution of the task at a specific time [1], [2], [3], [4], [5]. Utility functions capture the time-varying importance of a task to both the user and the enterprise as a whole. In this work, the value of completing a task decays over time and so we model monotonically-decreasing utility functions. The design of utility functions needs to be flexible to capture the importance of tasks within a diverse user base. In practice, utility functions may be created through a collaboration between the user and the owner of the computing system. We design dynamic resource management techniques to maximize the total utility that can be earned by completing tasks in an oversubscribed heterogeneous distributed environment. By oversubscribed we mean that the workload is large enough that the total desired work exceeds the capacity of the system in steady state operation, i.e., over an extended period. By a heterogeneous environment we mean that the execution time of each task may vary across the suite of machines. We model this computing environment and the workload of tasks that arrive dynamically. A scheduler makes resource allocation decisions to map (assign) the incoming tasks to the machines. The total utility earned from all completed tasks captures how much useful work was done and how timely that information was to the user. The system characteristics and the workload parameters are based on environments 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 both DOE and DoD environments.

We design a method that can be used to create utility functions by defining three parameters: priority, urgency, and utility class. The priority of a task represents the level of importance of a task to the enterprise, while urgency indicates how quickly the task loses utility. The utility class provides finer control of the shape of the utility function by partitioning it into intervals. We assume that the scheduler has experiential information about the execution time of each type of task on each type of machine. However, the scheduler does not know the arrival time, utility function, or type of each task until the task arrives.

- *B. Khemka, R. Friese, L.D. Briceño, H.J. Siegel, and A.A. Maciejewski are with the Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO 80526. E-mail: {Bhavesh.Khemka, Ryan.Friese, LDBricen, HJ, AAM}@colostate.edu.*
- *G.A. Koenig is with the Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN. E-mail: Koenig@ornl.gov.*
- *C. Groer is with Link Analytics, Atlanta, GA. E-mail: cgroer@gmail.com.*
- *G. Okonski, M.M. Hilton, and R. Rambharos are with the Department of Defense, Washington, DC. E-mail: {okonskitg, mmskizig}@verizon.net, Jendra.Rambharos@gmail.com.*
- *S. Poole is with the Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN and the DoD, Washington, DC. E-mail: swpoole@gmail.com.*

Manuscript received 24 Oct. 2013; revised 14 July 2014; accepted 4 Aug. 2014. Date of publication 25 Sept. 2014; date of current version 10 July 2015. Recommended for acceptance by D. Panda.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2014.2360513



(a) (b)

Fig. 1. (a) Four utility functions with different priority levels and a fixed urgency level showing the decay in utility for a task after its arrival time τ. The curves labeled "c," "h," "m," and "l" are the curves with *critical*, *high*, *medium*, and *low* priorities, respectively. (b) Four utility functions with different urgency levels and a fixed priority level showing the decay in utility for a task after its arrival time τ. The curves labeled "e," "h," "m," and "l" are the curves with *extreme*, *high*, *medium*, and *low* urgency levels, respectively. The length of time for which the starting utility value of a task persists (does not decay) is shorter for more urgent tasks.

We use two forms of dynamic heuristics for the resource allocation decisions. Immediate-mode heuristics schedule only the incoming task and do not have the opportunity to re-map tasks in machine queues (e.g., [6], [7], [8]). Batch-mode heuristics consider a set of tasks and have the ability to re-map tasks that are enqueued and waiting to execute (e.g., [6], [7]). We create seven immediate-mode and five batch-mode heuristics, and analyze their performance using simulation experiments. To examine the effect of oversubscription on the performance of the heuristics, we simulate two levels of oversubscription. We also study the effect of heuristic variations, such as dropping tasks and altering the mapping decision frequency for the batch-scheduler.

The contributions of this paper are: (a) a model of the planned DOE/DoD oversubscribed heterogeneous high performance computing environment, (b) the design of a metric using utility functions, based on the three parameters of priority, urgency, and utility class, to measure the performance of schedulers in an oversubscribed heterogeneous computing environment, (c) the design of twelve heuristics to perform the scheduling operations and their evaluation over a variety of environments, and (d) the exploration and the analysis of heuristic variations, such as dropping tasks and varying the number of tasks scheduled at each batch-mode mapping event.

The remainder of the paper is organized as follows. In Section 2, we explain our system model, including our method to design the utility functions (from the three parameters mentioned before), the characteristics of the workload, and the characteristics of the computing environment. We formally give our problem statement in Section 3, and introduce our metric to compare the performance of resource allocation heuristics. In Section 4, we describe the various heuristics we have designed and the method to drop tasks. We compare our study to other work from the literature in Section 5. We explore the design of our simulation experiments in Section 6. In Section 7, we present and analyze our simulation results. Finally, we conclude the paper and discuss possible future directions in Section 8.

2 System Model

2.1 Utility Functions

2.1.1 Overview

In our study, it is assumed that an enterprise computing system earns a certain amount of utility for completing each task. The amount of utility earned depends on the task and the time at which the task was completed relative to the time it arrived, and reflects its importance to the system. We use utility functions to model the time-varying benefit of completing the execution of a task. The utility functions we model are monotonically decreasing. This implies that if a task takes longer to complete, it cannot earn higher utility. We understand that there may be use cases for non-monotonically-decreasing utility functions, but they are not considered here. We design a flexible utility function for a task that is defined by three parameters: priority, urgency, and utility class. The goal is to use a small set of parameters that the users understand and enables the users to obtain the desired utility curve. By using a combination of these parameters we can create a variety of shapes for the utility functions. These parameters were designed based on the needs of the ESSC at ORNL. We expect that these parameters will be set by the customer (submitting the job) in collaboration with the system owner and the overall system administration policies.

2.1.2 Parameters

Priority. Priority represents the importance of the task to the organization. It sets the maximum value of a utility function. As the functions are monotonically decreasing, this is equivalent to the starting value of the utility function. Let $\pi(p)$ be the maximum utility (Max Util) of tasks belonging to priority level p, where $p \in \{critical, high, medium, low\}$. Each of these priority levels has a fixed value of maximum utility associated with it. Fig. 1a shows utility functions with different levels of priority for a fixed level of urgency (defined below). As shown in Fig. 1a, a task's utility does not begin to decay as soon as it arrives, because this would make the maximum utility value of a task unachievable (i.e., the task needs non-zero time to execute). In Section 6.2, we describe how we determine the length of this interval.

Urgency. The urgency of a task models the rate of decay of the utility of that task over time. It affects the "shape" of the utility function. Tasks that are more urgent will have their utility values decrease at a faster rate than less urgent tasks. In this study, we model the decay of utilities as an exponential (other functions may be used). Let $\rho(r)$ be the exponential decay rate of tasks belonging to urgency level r, where $r \in \{extreme, high, medium, low\}$. Fig. 1b illustrates utility functions with different urgency levels for a fixed priority level. The urgency level of a task along with the task's average execution time control the duration for which the starting utility value of a task does not decay (see Section 6.2).

Utility Class. A utility class is used to fine-tune a utility function by dividing the function into a set of intervals with discrete characteristics. We define each interval (except the first) to have three parameters: a start time, a percentage of maximum utility at that start time, and an exponential decay rate modifier. By defining different utility classes we can devise a wide variety of utility functions. We could set a hard deadline for a task by having the utility of the task drop to zero. For our simulations, we created four utility classes and each task belongs to one of these four classes (the number of utility classes can be domain dependent).

The first element within a utility class is the set of time intervals that partition the time axis of the utility function (except the end time of the first interval). Let $t(k, c)$ be the start time of the kth interval relative to the arrival time of a task belonging to utility class c.

The second element in a utility class sets the percentage of the maximum utility at the start of each of the intervals except the first. Let $\psi(k, c)$ be this percentage for the kth interval, where $0 \leq \psi(k, c) \leq 1$ and $\psi(k, c) \leq \psi(k-1, c)$ for $k > 1$. Therefore, the maximum utility value in the kth interval of a utility function for a task with a priority level p and utility class c is given by, $\Psi(k, c, p) = \psi(k, c) \times \pi(p)$.

The final element in a utility class c is a modifier, $\delta(k, c)$, to the exponential decay rate of the interval k, with $k > 1$ to ignore the first interval. The exponential decay rate in interval k of a utility function with urgency level r and utility class c is given by, $\Delta(k, c, r) = \delta(k, c) \times \rho(r)$. The values of this modifier are typically near 1, because the purpose of this modifier is to provide small differences in the decay rate across the intervals.

Fig. 2 shows a utility function (at a fixed priority and urgency level) partitioned into separate intervals, each with its own rate of decay and starting utility value. The last interval shows that the utility drops to zero as time tends to infinity.

2.1.3 Construction of a Utility Function

Let p be the priority level, r the urgency level, and c the utility class of a task i. The utility value $U(p, r, c, t)$ at any time t relative to the arrival time of the task, where $t(k, c) \leq t < t(k+1, c)$, is given by the following equation:

$$U(p, r, c, t) = (\Psi(k, c, p) - \Psi(k+1, c, p)) \times e^{-\Delta(k,c,r)\cdot(t-t(k,c))} + \Psi(k+1, c, p). \quad (1)$$

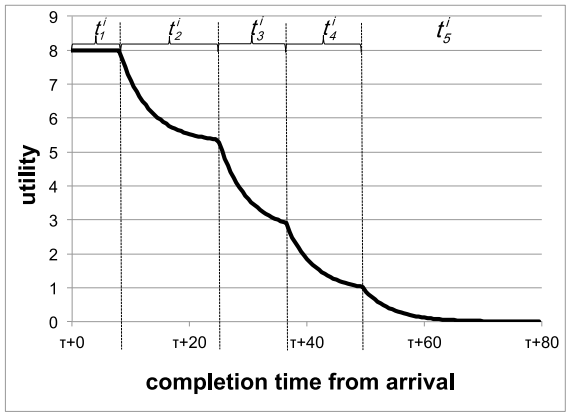


Fig. 2. Utility function for a fixed priority level, urgency level, and utility class, showing the decay in utility for a task after its arrival time τ. The t^is represent the duration of the different intervals in the utility class of task i. The last interval extends to infinity.

2.2 Model of Environment

We group tasks with similar computational requirements into task types and machines with similar performance capabilities into machine types. We model a heterogeneous environment, where the execution times of different task types may vary across the different machine types. We assume we are given an Estimated Time to Compute (ETC) matrix, where $ETC(i, j)$ is the estimated time to compute a task of type i on a machine of type j. This is a common assumption in the resource management literature [9], [10], [11], [12], [13], [14]. For simulation purposes, we use a synthetic workload as described in Section 6.3, but in practice, one could use historical data to obtain such information [11], [13]. We model special-purpose and general-purpose machines. The special-purpose machine types have the ability to execute certain task types much faster than the general-purpose machine types, but may be incapable of executing other task types. Further details are in Section 6.3.

We model a dynamic environment where tasks arrive throughout a 24 hour period. The scheduler does not know the arrival time, utility function, or type of each task until the task arrives. The system is composed of dedicated compute resources with a workload large enough to create an oversubscribed environment. We assume that the tasks in the workload are independent (no inter-task communication is required) and serial (each task executes on a single machine). For scheduling purposes we do not consider the pre-emption of tasks. We do, however, allow tasks to be dropped prior to execution (see Section 4.4).

3 Problem Statement

Our goal is to design resource management techniques to maximize the overall system utility achieved in an oversubscribed heterogeneous environment. To solve this problem, we devise twelve heuristics to perform the scheduling operations and design a metric using utility functions to measure the performance of schedulers. Once a task arrives, we can calculate the completion time of the task based on the resource to which it is mapped. Using the completion time of task i, denoted $t_{completion}(i)$, and the task's utility function parameters (namely, $p(i), r(i)$, and $c(i)$), the utility earned by the task can be calculated using Equation (1) to obtain

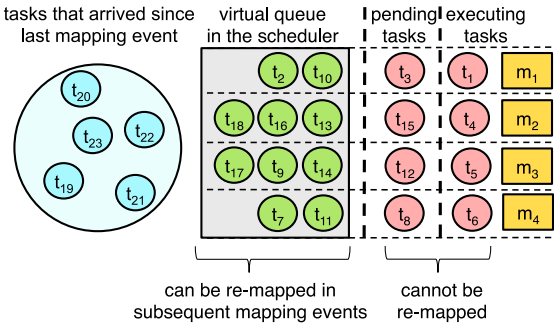


Fig. 3. Machine queues of a sample system with four machines. The tasks in the executing and pending slots are not eligible to be re-mapped, whereas the tasks in the virtual queue section of the machine queues can be re-mapped. This only applies to the batch-mode heuristics.

$U(p(i), r(i), c(i), t_{completion}(i))$. Let $\Omega(t_{end})$ be the set of tasks that have completed execution by time t_{end}. The goal of our resource management procedures is to maximize the total utility that can be earned by the system over the 24 hour period and is computed using:

$$U_{system}(t_{end}) = \sum_{i \in \Omega(t_{end})} U(p(i), r(i), c(i), t_{completion}(i)). \quad (2)$$

4 Resource Management Policies

4.1 Overview

The scheduling problem, in general, has been shown to be NP-complete [15], and therefore it is common to use heuristics to solve this problem. Any time when a decision has to be made to assign a task to a machine we call a mapping event. The two types of dynamic heuristics (also known as online heuristics [16]) we use, immediate-mode heuristics and batch-mode heuristics, differ in the method that a mapping event is triggered and in the set of tasks that can be scheduled during a mapping event.

In immediate-mode heuristics, a mapping event occurs when a task arrives. The only exception to this is when the execution of the previous mapping event has not finished before the arrival of the next task. In that case, the trigger time for the next mapping event is delayed until the previous mapping event finishes execution. The immediate-mode heuristics assign the new task to some machine queue. Once the task is put in the machine queue it cannot be remapped. We design and evaluate seven immediate-mode heuristics.

In batch-mode heuristics, typically mapping events are triggered after fixed time intervals or a fixed number of task arrivals. If the previous mapping event has not completed execution, the trigger time of the next mapping event is delayed until the previous mapping event finishes execution. We refer to the task that is next in-line for execution on a machine queue as a pending task. We refer to the part of the machine queues that do not include the executing and the pending tasks as the virtual queues of the scheduler. Fig. 3 shows tasks waiting in the virtual queues of an example system with four machines. The batch-mode heuristics make mapping decisions for both the tasks that have arrived since the last mapping event and the tasks that are waiting in the virtual queues. This set of tasks is the called the mappable tasks. The batch-mode heuristics (unlike immediate-mode heuristics) have the capability to re-map tasks in the virtual queues of scheduler. The batch-mode heuristics do not re-map pending tasks so the machine does not become idle when its executing task completes. The simulation results in Section 7 show that the batch-mode heuristics have a significant advantage because they have more information available while making a mapping decision (as they consider a set of tasks). Furthermore they can alter those decisions in the future by remapping tasks when additional information becomes available. We design and evaluate five batch-mode heuristics.

4.2 Immediate-Mode Heuristics

4.2.1 Naive Immediate-Mode Heuristics

The first two immediate-mode heuristics do not consider the execution time estimates of different task types on machine types, nor the ready-times of the machines (the times that the machines finish execution of their already queued tasks). These heuristics are used as baseline heuristics for comparison purposes. We refer to these heuristics as the naive immediate-mode heuristics.

The Random heuristic assigns the newly arrived task to a random machine on which it can execute (i.e., not a special-purpose machine where it cannot execute). The Round-Robin heuristic assigns the incoming tasks in a round-robin fashion. The machines are listed in a randomized order and this ordering is kept fixed. The first task that arrives for a mapping event is assigned to the first machine (on which it can execute), the next incoming task is assigned to the next machine (on which it can execute), and so on.

4.2.2 Smart Immdediate-Mode Heuristics

We refer to the next five heuristics as the smart immediate-mode heuristics. The results in Section 7 show that these heuristics perform better than the naive immediate-mode heuristics.

The Maximum Utility heuristic is based on the Minimum Completion Time heuristic from the literature [7], [17], [18], [19], [20]. The heuristic assigns a newly arrived task to the machine that would complete it soonest. We model monotonically-decreasing utility functions and, thus, the machine that completes the task the earliest is also the machine that earns the highest utility from the task. This heuristic accounts not only for the execution time of the task on machines, but also the ready-time of the machines.

The Maximum Utility-Per-Time (Max UPT) heuristic computes the utility a newly arrived task can earn on each machine divided by its execution time on that machine. It then assigns the task to the machine that maximizes "utility earned / execution time." The reasoning behind this is to earn highest utility per time in an oversubscribed system.

We design two heuristics based on the Minimum Execution Time (MET) heuristic [7], [17], [18]. The Minimum Execution Time-Random (MET-Random) heuristic first finds the set of machines that belong to the machine type that can *execute* the newly-arrived task the fastest (ignoring machine ready time). Among those machines, it assigns the task to a random machine. The Minimum Execution Time-Max Util (MET-Max Util) heuristic also finds

the set of machines belonging to the minimum *execution* time machine type for the newly arrived task, but picks the machine among them that minimizes *completion* time (which also maximizes utility).

The K-Best Types heuristic is based on the K-Percent Best heuristic, introduced in [7] and used in [18], [21], [22], [23]. The idea is to try combining the benefits of the MET heuristic and the Max Util heuristic. The K-Best Types heuristic first finds the K-best machine types that have the lowest *execution* times for the current task. Among the machines of these machine types, it then picks the machine that minimizes *completion* time (which also maximizes utility). By using different values of K, we can control the extent to which the heuristic is biased towards MET-Max Util or Max Util. We empirically determine the best value of K.

4.3 Batch-Mode Heuristics

The Min-Min Completion Time (Min-Min Comp) heuristic is based on the concept of the two-stage Min-Min heuristic that has been widely used (e.g., [6], [7], [18], [19], [20], [22], [23], [24], [25], [26], [27]). In the first stage, the heuristic independently finds for each mappable task the machine that can complete it the soonest. In the second stage, the heuristic picks from all the task-machine pairs (of the first stage) the pair that has the earliest completion time. The heuristic assigns the task to that machine, removes that task from the set of mappable tasks, updates the ready-time of that machine, and repeats this process iteratively until all tasks are mapped. This batch-mode heuristic is computationally efficient because it explicitly does not perform any utility calculations.

The Sufferage heuristic concept introduced in [7] and used in, for example, [6], [19], [22], [24], [25], [28], [29], attempts to assign tasks to their maximum utility machine. Ties are broken in favor of the tasks that would "suffer" the most if they did not get their maximum utility machine. In the first stage, the heuristic calculates for each mappable task a sufferage value, i.e., the difference between the best and the second-best utility values that the task could possibly earn. In the second stage, tasks are assigned to their maximum utility machines. If multiple tasks request the same machine, then the task that has the highest sufferage value is assigned to that machine. Assigned tasks are removed from the mappable tasks set, ready-times of machines updated, and the process repeated until all tasks are mapped.

The Max-Max Utility (Max-Max Util) heuristic is also a two-stage heuristic, like the Min-Min Comp heuristic. The difference is that in each stage Max-Max Util maximizes utility, as opposed to minimizing completion time. In the first stage, this heuristic finds task-machine pairs that are identical to those found in the first stage of the Min-Min Comp heuristic, because of the monotonically-decreasing utility functions. In the second stage, the decisions made by Max-Max Util may differ from those of Min-Min Comp. This is because in the second stage, the Max-Max Util heuristic picks the maximum utility choice among the different task-machine pairs, and the utility earned depends both on the completion time and the task's specific utility function.

The Max-Max Utility-Per-Time (Max-Max UPT) heuristic is similar to the Max-Max Util heuristic. The difference being that in each stage Max-Max UPT maximizes "utility earned / execution time," as opposed to maximizing utility. As mentioned before, this heuristic attempts to maximize utility earned by a task while minimizing the time it uses computational resources. Completing tasks sooner is helpful in an oversubscribed system.

The MET-Max Util-Max UPT heuristic is similar to the Max-Max UPT heuristic with a difference in the first stage. In the first stage, this heuristic pairs each task with the minimum completion time machine among the machines that belong to its minimum execution time machine type. Therefore, for a task, this batch-mode heuristic performs utility calculations only for a subset of the machines (i.e., those machines that belong to the machine type that executes this task the fastest).

4.4 Dropping Low-Utility Tasks

In an oversubscribed environment, it is not possible to earn significant utility from all tasks. We introduce the ability to drop low-utility earning tasks while making mapping decisions. Dropping a task means that it will never be mapped to any machine (unless the user resubmits it). The motivation for doing this is to reduce the wait times (i.e., increase the achieved utility) of the other (higher-utility earning) tasks that are queued in the system. In practice, we expect that policy decisions will determine the extent to which this technique is applied, and that it will only be used in extreme situations. The extent of dropping is a tunable parameter that can be varied based on the system oversubscription level. The goal is to drop tasks that would earn less utility than a pre-set threshold, referred to as the dropping threshold. In this study, for each simulation, the dropping threshold is fixed at a particular value. The model can be extended to have a dropping threshold that varies based on the current or expected system load. We use different methods to drop tasks in the immediate-mode and the batch-mode heuristics.

For the immediate-mode heuristics, the decision to drop a task is made after the heuristic determines the machine queue in which to map the task. We can compute the completion time of the task on this machine and the utility that this task will earn. If the utility earned by this task is less than the dropping threshold, we do not assign the task to the machine, and drop it from any further consideration. If the utility earned is greater than or equal to the dropping threshold, the task is placed on the machine queue as decided by the heuristic.

For the batch-mode heuristics, the decision to drop a task requires more computation because of the possibility of the task being remapped to another machine in a subsequent mapping event. Before calling the heuristic, for each mappable task, we determine the maximum possible utility that the task could earn on any machine assuming it could start execution immediately after the pending task. If this utility is less than the dropping threshold, we drop this task from the set of mappable tasks. If it is not less than the threshold, the task continues to stay in the set of mappable tasks and the batch-mode heuristic performs its allocation decisions.

In addition to the dropping operation, for the batch-mode heuristics, we implement a technique to permute tasks that are at the head of the virtual queues of the machines, but this did not improve performance. This

technique is described in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2014.2360513.

5 Related Work

Numerous studies have proposed heuristics to solve the problem of performing resource management in dynamic heterogeneous computing environments (e.g., [6], [7], [22], [23]). Few of them, however, optimize for the total utility that the system earns. In a survey of utility function based resource management [2], the authors point out that in an oversubscribed system it is preferable to use utility accrual algorithms for performing scheduling decisions because these have the ability to pick and execute tasks that are more important to the system (earn high utility). Additional research explores developing a framework for measuring the productivity of supercomputers [30]. They propose a metric for productivity that is the ratio of the utility earned by completing a task to the cost of doing this operation. Possible shapes for the utility-time and the cost-time curves of a task are discussed. The authors also mention the possible interpretations of "utility" and "cost." Similar to our work, they consider only monotonically-decreasing utility-time functions. Our work enhances this by parameterizing the shape of the utility functions and designing resource management techniques to maximize the aggregate utility.

Value functions (similar to utility functions) are used in systems with processes running on symmetric, shared-memory multi-processors (SMP) with one to four processing elements [1]. Each process has a value function associated with it that specifies the value earned by the system depending on when it completes execution of that process. The scheduler can consider the arrival times of tasks to make current scheduling decisions. Moreover, the processes can be periodic. This is in contrast to our model where the scheduler has no prior knowledge of the arrival time of the tasks. The paper presents two algorithms that make decisions based on value density (value divided by processing time) and shows that these algorithms perform better than scheduling algorithms that consider either only deadlines or only execution times (ignoring the utility earned). This is similar to some of our heuristics that use utility-per-time. Unlike our environment, they consider homogeneous processing elements. Other systems using similar value functions have also been examined [31], [32].

Kim et al. define tasks with three soft deadlines [22]. The actual completion time of the task is compared to the soft deadlines to obtain a deadline factor. The deadline factor is multiplied with the priority of a task to calculate the actual "value" that is earned for completing the task. Dynamic heuristics are used to maximize the total value that can be earned by mapping the tasks to machines. Although tasks can have different priorities, the degradation curve for the value of a task is always a step-curve with the steps occurring at the soft deadlines. In our model, each task can have its own utility function shape and the utility decays exponentially. Also, we model special-purpose machine and task types, have different arrival patterns for the different kinds of tasks, and experiment with dropping low utility-earning tasks in our oversubscribed system.

The concepts of utility functions have been used in real-time systems for scheduling tasks [3], [4]. The problem of scheduling non-preemptive and mutually independent tasks on a single processor has been examined [3]. In that study, each task has a time value function that gives the task's contribution at its completion time. The goal is to order the execution of the tasks on the single processor to maximize the cumulative contribution from the tasks. Analytical methods have been used to create performance features and optimize them [4]. In that study, all jobs have the same shape for their utility functions, as opposed to our study where every task can have a different shape for its utility function. Although these papers address the maximization of total utility earned, the environment of a single processor versus our environment of a heterogeneous distributed system makes solution techniques significantly different for the two cases.

In [5], the users of a homogeneous high performance computing system can draw arbitrary shapes for utility functions for the jobs they submit. The users decide the level of accuracy in modeling the utility functions. The work in [5] uses a genetic algorithm to solve the problem of maximizing utility. The average execution time of the algorithm is 8,900 seconds. In our study, scheduling decisions are made at much smaller intervals (after a minute in the case of the batch-mode heuristics). Furthermore, we assume a heterogeneous computing system, as opposed to the homogeneous computing system that they model.

6 Simulation Setup

6.1 Overview

In this study, we simulate a heterogeneous computing environment where a workload of tasks arrive dynamically. To model the execution time characteristics of the workload, we use an Estimated Time to Compute matrix (as described in Section 2.2). To completely describe the workload, we need to determine each task's utility function parameters, task type, and arrival time. In this section, we explain how we generate these parameters for our simulations *based on the expectations for future environments of DOE and DoD interest*.

Each experiment discussed in Section 7 has its results averaged over 50 simulation trials. Each trial has a new workload of tasks (with different utility functions, task types, and arrival times). Each trial models a different compute environment by using different values for the entries of the ETC matrix. We now describe our method of generating these values for each of the trials.

6.2 Generating Utility Functions

For each task in the workload, we need to assign the three parameters to describe its utility function (i.e., priority, urgency, and utility class). As mentioned in Section 2.1.2, we have four possibilities for each of these parameters. We model four utility classes in this study because these are representative of the expected workload at ESSC. In our simulations, a task's utility class is chosen uniformly at random among the four classes modeled. Fig. 4 illustrates the utility functions obtained by using the four utility classes that we used in this study for a fixed priority level and a

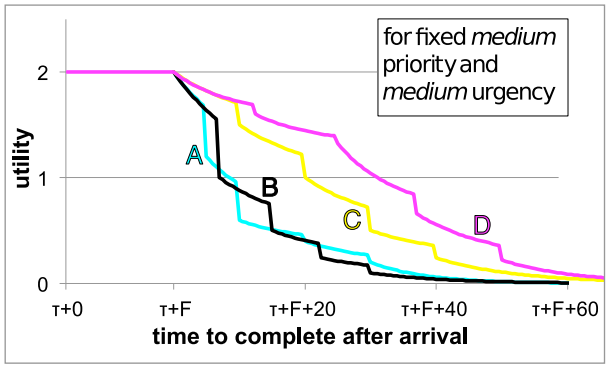


Fig. 4. The utility functions of the four utility classes (A, B, C, and D) used in this study shown at fixed priority and urgency levels showing the decay in utility for a task after its arrival time τ. The duration of its first interval during which the utility value remains constant is represented by F on the x-axis.

fixed urgency level. The length of the first interval during which the utility value does not decay is represented by "F" in the figure. It is dependent on the urgency level of the task as well as the average execution time of the task. Appendix B, available in the online supplemental material, gives the method for its computation. Appendix C, available in the online supplemental material, gives the values used to create the four utility classes.

In our simulations, the priority and urgency levels of a task are set based on a joint probability distribution that is representative of DOE/DoD environments. Appendix D, available in the online supplemental material, shows this probability distribution as a matrix. The model results in most tasks having *medium* and *low* priorities with *medium* and *low* urgencies, and a few important tasks having *critical* and *high* priorities with *extreme* and *high* urgencies.

The values of maximum utility set by the various priority levels are: $\pi(critical) = 8$, $\pi(high) = 4$, $\pi(medium) = 2$, and $\pi(low) = 1$. We also experimented with a different set of values for the priority levels: $\pi(critical) = 1{,}000$, $\pi(high) = 100$, $\pi(medium) = 10$, and $\pi(low) = 1$. The exponential decay rates for the various urgency levels are: $\rho(extreme) = 0.6$, $\rho(high) = 0.2$, $\rho(medium) = 0.1$, and $\rho(low) = 0.01$. These priority and urgency values are based on the needs of the ESSC.

6.3 Generating Estimated Time to Compute Matrices

In our simulation environment, we group together tasks that have similar execution time characteristics into task types, and machines that have similar performance capabilities into machine types. We model 100 task types and 13 machine types. In our simulations, the procedure by which we assign tasks to task types is described in Section 6.4. We model an environment consisting of 100 machines, where each machine belongs to one of 13 machine types. Among these 13 machine types, four are special-purpose machine types while the remaining are general-purpose machine types. We model the special-purpose machine types as having the capability of executing certain task types (which are special to them) approximately ten times faster than on the general-purpose machine types. These special-purpose machine types, however, lack the ability to execute the other task types. In our environment, three to five task types were special on each special-purpose machine type.

We use techniques from the Coefficient of Variation (COV) method [33] to generate the entries of the ETC matrix. The mean value of execution time on the general-purpose and the special-purpose machine types is set to 10 minutes and one minute, respectively. Complete details about our parameters for generating ETC matrices are described in Appendix E, available in the online supplemental material. The appendix also discusses how we distribute the 100 machines among the 13 machine types.

In this study, the task type of a task is not correlated to the worth of the task to the system, and therefore is not related to the utility function of the task. The task type only controls the execution time characteristics of the task.

6.4 Generating the Arrival Pattern of Tasks

To generate the arrival times of the tasks in the simulation, we use different arrival patterns for the special-purpose and the general-purpose task types. The goal of our arrival pattern generation is to closely model expected workloads of DOE and DoD interest. Our simulation models the arrival and mapping of tasks for a 24 hour period. Real-world oversubscribed systems rarely start with empty queues, so we simulate the arrival and mapping of tasks for 26 hours, and exclude the first two hours of data from result calculations. The initial two hours serve to bring the system up to steady-state. We calculate all of our results (utility earned, average heuristic execution time, number of dropped tasks, etc.) for the duration of 24 hours (i.e., from the end of the 2^{nd} to the end of the 26^{th} hour). Based on the estimated level of oversubscription for the DOE/DoD environments of future interest, we simulated 33,000 task arrivals during a 24 hour period. To examine the impact on performance if the system were extremely oversubscribed, we also experimented with 50,000 tasks arriving in a day.

Before we generate the arrival patterns for the special-purpose and the general-purpose tasks types, we first find a mean arrival rate of tasks for every task type (irrespective of special-purpose or general-purpose). We find the estimated number of tasks of each task type that will arrive during the day by sampling from a Gaussian distribution. The mean for this distribution is the ratio of the desired number of tasks to arrive (33,000 or 50,000) to the number of task types in the system. The variance is set to $1/10^{th}$ of the mean. We obtain the mean arrival rate of a task type by dividing the estimated number of tasks of this task type that are to arrive during the period by 24 hours. The mean arrival rate of each task type is used to generate arrival rate patterns (that have different arrival rates during the 24 hours), based on whether it is a special-purpose or a general-purpose task type. For the general-purpose task types, we use a sinusoidal pattern for the arrival rate. For the special-purpose task types, we use a bursty arrival rate pattern. Appendix E, available in the online supplemental material, discusses how we create the arrival pattern for a general-purpose or special-purpose task type, and use this arrival rate pattern of a task type to obtain the actual number and arrival times of the tasks belonging to that task type.

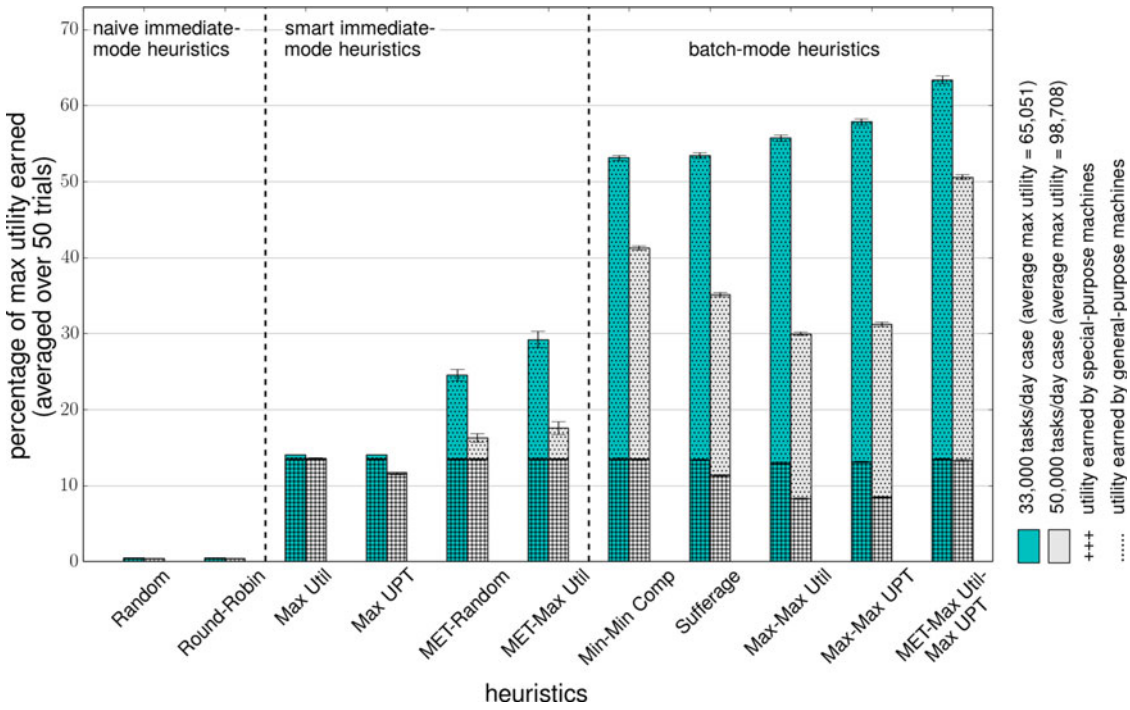


Fig. 5. Percentage of maximum utility earned by all the heuristics under two levels of oversubscription: 33,000 tasks arriving within a day, and 50,000 tasks arriving within a day. No tasks were dropped in these cases. The utility earned value (as opposed to the percentage of maximum utility earned) by a heuristic in the 50,000 tasks per day case will typically be higher than that in the 33,000 tasks per day case.

7 Simulation Results and Analysis

7.1 Overview

As mentioned in the previous section, we generate 50 simulation trials for each experiment that we describe in this section. All bar charts in this section have results averaged over the 50 trials with error bars showing 95 percent confidence intervals. For the batch-mode heuristics, the next mapping event occurs after both of the following conditions have been met: a time interval of one minute has passed since the last mapping event and the execution of the previous mapping event has finished. Later in this section, we show results with different methods of triggering batch-mode mapping events.

To make a fair comparison across the two levels of oversubscription, it is important to analyze the performance of a heuristic as a percentage of the maximum possible utility that could be achieved in that oversubscription level. The value of maximum utility bound that can be earned is calculated by summing the utility values achieved if all tasks were assumed to begin execution on their minimum execution time machine as soon as they arrive. We consider only tasks whose completion times are within the 24 hour period. The values of the maximum utility bound averaged across the 50 trials in the 33,000 and 50,000 tasks arriving per day cases are 65,051 and 98,708, respectively. First, we compare the performance of the various heuristics with the two levels of oversubscription. We then explore the effect of dropping tasks with different levels of dropping thresholds.

The best value of K for the K-Best Types heuristic was empirically found to be K=1 machine type in our environment. At K=1, the K-Best Types performs the same mapping decisions as the MET-Max Util heuristic. We therefore do not show the results from this heuristic.

7.2 Preliminary Results

Fig. 5 shows the performance of the different heuristics in terms of the percentage of maximum utility earned with the two levels of oversubscription. Irrespective of the oversubscription level, we observe that the naive immediate-mode heuristics always perform poorly compared to the smart immediate-mode heuristics. This is because the naive heuristics do not consider ETC information, machine ready-times, and the utility earned by a task on the various machines. The batch-mode heuristics always perform significantly better than the smart immediate-mode heuristics. This is because the batch-mode heuristics not only consider machine ready-times, but also have the ability to schedule a set of tasks and re-map tasks that are in the virtual queues. Most of the batch-mode heuristics are able to use this to their advantage and move any high utility-earning task that may have just arrived to the front of the virtual queues in the next mapping event. With the immediate-mode heuristics, the newly-arrived high utility-earning tasks would be queued behind other tasks, and by the time they get an opportunity to execute, their utility may have decayed significantly. With the 33,000 tasks per day case, on average, the batch-mode heuristics gave an improvement of approximately 250 percent compared to the smart immediate-mode heuristics.

Comparing the percentage of maximum utility earned by the heuristics for the two levels of oversubscription shows that higher oversubscription makes it harder to earn the maximum possible utility. The actual utility earned by a heuristic in the 50,000 tasks per day case will typically be higher than that in the 33,000 tasks per day case. For example, the utility earned by Min-Min Comp in the 33,000 tasks per day case is 53.13 percent of 65,051 = 34,555, and in the 50,000 tasks per day case is 41.26 percent of 98,708 = 40,726.

Even though for both levels of oversubscription we consider the utility earned by the system only for the 24 hour duration, the higher oversubscription rate allows a heuristic to select more higher utility earning tasks, and therefore earn higher utility.

The Max Util and Max UPT immediate-mode heuristics earn most of their utility from the special-purpose machines. This is because the special-purpose machines are able to quickly execute the tasks assigned to them (i.e., special-purpose tasks) and these machines are not oversubscribed. As a result, a task assigned to a special-purpose machine begins execution quickly and is able to earn high utility. In contrast, the general-purpose machines have long queues of tasks and therefore the tasks assigned to them usually earn very low utility by the time they finish execution. MET-Random and MET-Max Util alleviate this problem by assigning tasks to machines where they execute the fastest. This allows these heuristics to earn utility from the general-purpose machines as well.

The performance of many batch-mode heuristics is severely affected by the increase in the oversubscription level. The higher oversubscription results in more tasks being present in the batch during the mapping events. With an increase in the size of the batch, the batch-mode heuristics take considerably longer to perform each mapping event. This leads to triggering fewer mapping events (because a new mapping event cannot begin until the previous one completes). Fig. 7 shows the total number of mapping events for the batch-mode heuristics under the two levels of oversubscription. The total number of mapping events are partitioned into two sections: those triggered at the time interval of one minute and those initiated when the execution of the previous mapping event took longer than one minute. We observe that the batch-mode heuristics (other than Min-Min Comp in 33,000 tasks per day case) have fewer mapping events being triggered than the expected amount (namely, 1,440 if they were all triggered after one minute). With fewer mapping events, it takes longer for the high utility-earning tasks to be moved up to the front of the virtual queues and the delay may cause their utility values to decay significantly. Min-Min Comp executes faster than the other batch-mode heuristics because it does not perform any explicit utility calculations. MET-Max Util-Max UPT also executes relatively quickly because it performs utility calculations only for a subset of the machines. Max-Max Util and Max-Max UPT earn very low utility in the 50,000 tasks per day case because they have only 200 mapping events being triggered during the day. In contrast to the batch-mode heuristics, the immediate-mode heuristics execute quickly, and as a result, even in the case where 50,000 tasks arrive during the day, they have approximately 50,000 mapping events with only 0.5 percent of those on average (250 out of 50,000) being initiated as a result of the heuristic execution of the previous mapping event taking longer than the arrival time of the next task.

Picking the minimum execution time machine type for a task is automatically providing load balancing in our environment. The MET-type heuristics (both immediate-mode and batch-mode) are performing particularly well because of the high heterogeneity modeled in our environment. If we had a variation in our environment where the workload includes many task types that perform best on a select few machines, these MET-type heuristics would assign all of those tasks only to these few machines resulting in long machine queues on these fast machines, where the wait time of a task would negate the faster execution time. Our level of heterogeneity is modeled based on the expectations for future environments of DOE and DoD interest.

7.3 Results with Dropping Tasks

As mentioned in Section 4.4, we implement techniques in the immediate-mode and batch-mode heuristics to drop tasks that earn utility values less than a dropping threshold. We experiment with six levels for the dropping threshold: 0 (which is equivalent to no dropping), 0.05, 0.5, 1.5, 3, and 5. These are chosen based on our system model, including the values of maximum utility for the various priority levels, i.e., 8, 4, 2, and 1. We run simulations with all the heuristics using the six dropping thresholds for the two cases of oversubscription. In Fig. 6, we show the results for the 50,000 tasks per day case. The results of the 33,000 oversubscription level show similar trends, and are discussed in Appendix G, available in the online supplemental material. The heuristics significantly benefit from the dropping operation. For almost all heuristics, the utility earned increases as we increase our dropping threshold from 0 to 1.5. With a dropping threshold of 1.5, all the *low* priority tasks are dropped because their starting utility is 1. This may be undesirable in general, but for our oversubscribed system this results in the best performance. The average computation capability of our environment is such that approximately 26,000 tasks can execute in the 24-hour period (based on the average execution time of each task on each machine). Our dropping operation lets us pick the best 26,000 tasks to execute to maximize the total utility that can be earned. Based on a different system model and administrative policies one may set the specific levels of dropping thresholds differently.

The immediate-mode heuristics do not have the ability to move newly arrived high-utility earning tasks to the head of the queue because they are not allowed to remap queued tasks. The dropping operation benefits the immediate-mode heuristics by clearing the machine queues of the lower-utility-earning tasks, which allows the other queued tasks to execute sooner and earn higher utility. This helps the immediate-mode heuristics to earn utility from the general-purpose machines. The special-purpose machines were not oversubscribed and therefore there is no significant increase in performance from these machines because of the dropping operation. At the best dropping threshold, i.e., 1.5, Max Util and Max UPT have an approximately 450% performance improvement compared to the no dropping case. The performance of these two heuristics becomes comparable to that of the batch-mode heuristics. As we increase the dropping threshold beyond 1.5, we drop too many tasks from our system and earn less utility overall.

There are two main reasons why the batch-mode heuristics benefit from the dropping operation. The first is that the dropping operation helps them reduce the size of their batch during each mapping event by dropping tasks that would only be able to earn low utility. This makes the mapping events execute faster and results in more mapping

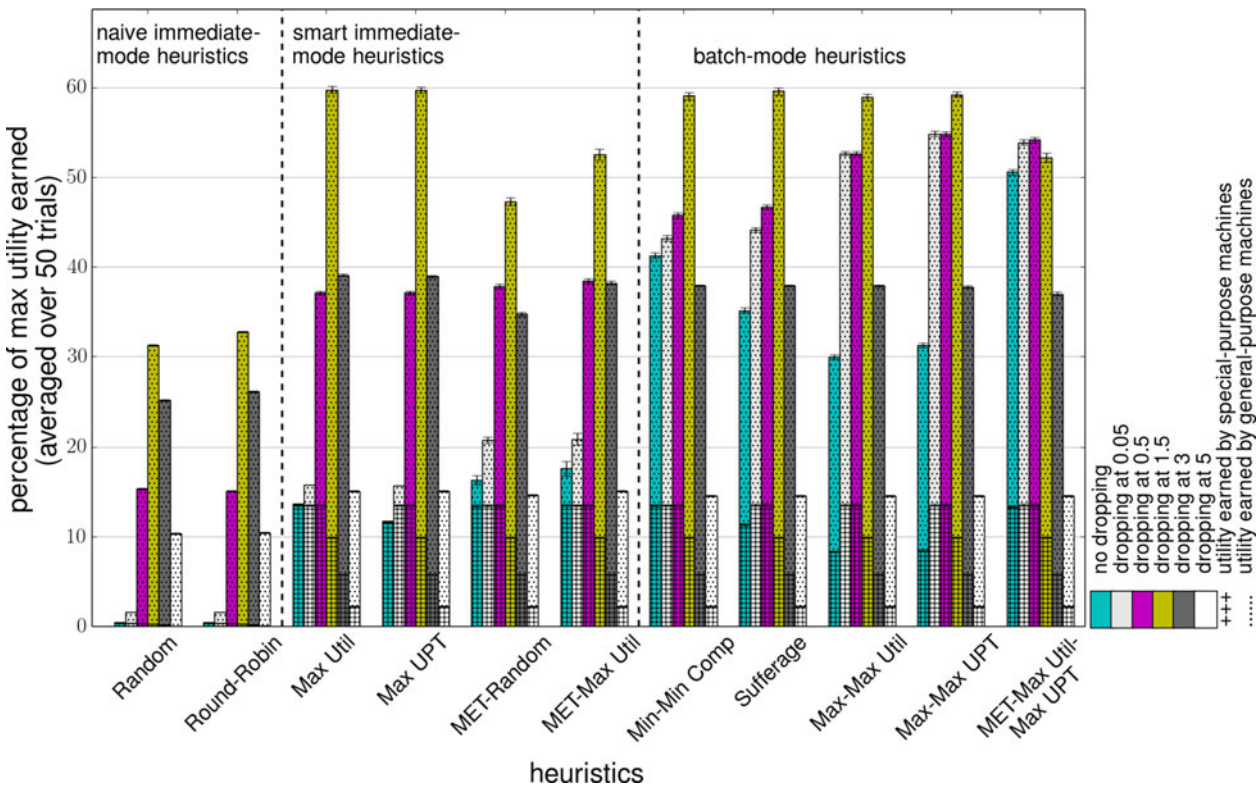


Fig. 6. Percentage of maximum utility earned earned by all the heuristics for the different dropping thresholds with the oversubscription level of 50,000 tasks arriving during the day. The average maximum utility bound for this oversubscription level is 98,708.

events. For the batch-mode heuristics, in all cases where some level of dropping was implemented, all of the 1440 mapping events were triggered. With the increase in the number of mapping events, the batch-mode heuristics are able to service high utility-earning tasks faster. This causes the improvement in performance in the dropping at 0.05 case compared to the no dropping case. The second reason the batch-mode heuristics benefit from the dropping operation is the prevention of low utility-earning tasks from blocking the pending and the executing slots of the machines. When tasks are arriving, there may be periods when most of the arriving tasks are neither *critical* nor *high* priority tasks. During this time period, other lower priority tasks get the opportunity to fill into the pending slots of the machines. If there is a burst of *critical* or *high* priority tasks after this period, these higher-priority tasks will have to wait in queue behind the lower priority task in the pending slot, because the pending slot tasks cannot be re-mapped. By dropping the lower priority tasks, we do not block the pending (and hence the executing) slots and when the high utility-earning tasks arrive they get to quickly start execution and provide higher utility to the system. This causes the performance improvement for batch-mode heuristics with further dropping beyond 0.05. Similar to the immediate-mode

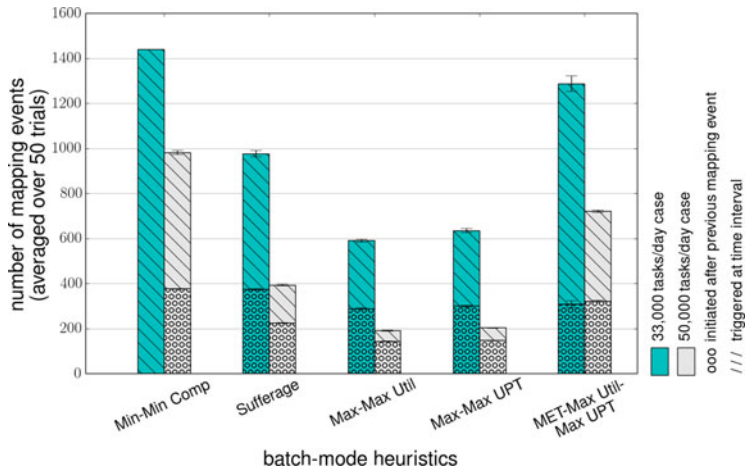


Fig. 7. The number of mapping events initiated either because the one minute time interval has passed since the last mapping event or because the previous mapping event finished execution after one minute are shown for five batch-mode heuristics with the two levels of oversubscription: 33,000 tasks arriving within a day, and 50,000 tasks arriving within a day. No tasks were dropped in these cases.

heuristics, dropping thresholds greater than 1.5 drop too many tasks.

Max-Max Util, Max-Max UPT, and MET-Max Util-Max UPT maximize the utility earned and push low utility-earning tasks to the back of the queue. Thus, for these heuristics, the main advantage of dropping is to reduce the size of the batch, allowing more mapping events.

The dropping operation also helps to make the Min-Min Comp and Sufferage heuristics more utility-aware, and we get the biggest performance improvement by increasing the dropping threshold to 1.5 (even though the dropping threshold at 0.05 triggered all 1,440 mapping events).

In all cases where some level of dropping is implemented, almost all of the heuristics earn similar values of utility from the special-purpose machines because these machines are not oversubscribed. Utility earned from special-purpose machines decreases with dropping thresholds of 1.5 and higher because proportionally the number of special-purpose tasks become fewer. As the special-purpose machines are not oversubscribed, one could get higher performance from the system by having two separate dropping thresholds: one for special-purpose tasks and the other for general-purpose tasks.

Although the smart immediate-mode heuristics can earn utility comparable to the batch-mode heuristics, their performance is very sensitive to the value of the dropping threshold. For the immediate-mode heuristics, the dropping threshold parameter needs to be tuned based on the starting utility values for the different priority levels, arrival pattern of the tasks, degree of oversubscription of the environment, etc., because the immediate-mode heuristics rely on the dropping threshold to empty the machine queues. In contrast, the mechanism by which the dropping operation helps batch-mode heuristics such as Max-Max Util, Max-Max UPT, and MET-Max Util-Max UPT is different, i.e., it increases the number of mapping events. The performance of these batch-mode heuristics is less sensitive to the value of the dropping threshold.

The MET-based heuristics, i.e., MET-Random, MET-Max Util, and MET-Max Util-Max UPT, earn less utility compared to the other heuristics at a dropping threshold of 1.5. At this dropping threshold, all the *low* priority tasks are dropped from the system, and they account for approximately 53 percent of tasks (see Appendix D, available in the online supplemental material). Therefore, with a 1.5 dropping threshold the degree of oversubscription reduces significantly. The MET-based heuristics assign tasks to the machines that belong to the best execution time machine type. As a result, these heuristics hurt their case at this dropping threshold by oversubscribing certain machines. This causes them to drop more tasks (because tasks wait longer) compared to the other heuristics and earn less utility overall. The effect of increased oversubscription by the MET-based heuristics is not apparent at the 0.5 and 3 dropping thresholds because at these dropping thresholds the system is much more oversubscribed and undersubscribed, respectively.

For dropping thresholds 1.5 and above, most heuristics earn similar amounts of total utility (except naive heuristics and the MET-based heuristics). At these dropping thresholds, only tasks of higher priority levels are executing on the machines (as tasks with lower priority levels have starting utility values less than the dropping threshold) and as a result the degree of oversubscription is reduced. The non-dropped tasks start execution as soon as they arrive because machines are idle most of the time, and therefore, all heuristics earn similar levels of utility.

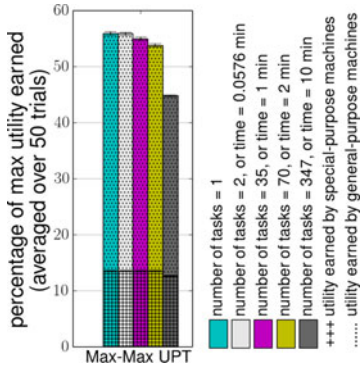


Fig. 8. Percentage of maximum utility earned by the Max-Max UPT heuristic for the different cases of triggering batch-mode mapping events. The other batch-mode heuristics show similar trends.

The average mapping event execution times for the heuristics in both levels of oversubscription at a 0.5 dropping threshold are in Appendix G, available in the online supplemental material. Results of experiments with the maximum utility values for the priority levels set at 1000, 100, 10, and 1 are discussed in Appendix H, available in the online supplemental material.

7.4 Triggering Batch-Mode Mapping Events

The ability of the batch-mode heuristics to update the machine queues with a high utility-earning task that may have arrived recently provides a distinct advantage. We now study the effect of varying the size of the batch by exploring other possibilities for triggering the next mapping event. We examine a technique to trigger batch-mode mapping events based on a combination of time interval and number of tasks that have arrived since the last mapping event. A mapping event will be triggered when either of the above (time interval or number of tasks) occur, or after the previous mapping execution if it takes longer. These studies are performed using the 0.5 dropping threshold and 50,000 tasks per day case. We experiment with the following five triggering cases: (1) number of tasks: 1; (2) number of tasks: 2, or time interval: 0.0576 minutes; (3) number of tasks: 35, or time interval: 1 minute; (4) number of tasks: 70, or time interval: 2 minutes; (5) number of tasks: 347, or time interval: 10 minutes. For each case, the time intervals are chosen to approximate the corresponding estimated number of task arrivals. These experiment parameters are set based on our simulation environment. One could perform such tests with different values for the parameters based on other environments.

Fig. 8 shows the performance of the Max-Max UPT heuristic with the different cases of triggering. The other batch-mode heuristics show similar trends as the Max-Max UPT heuristic. In all five triggering cases mentioned above and for all of the batch-mode heuristics, the average execution time of a mapping event with a dropping threshold of 0.5 is under 350 milliseconds.

The best performance is obtained when mapping events trigger every time a new task arrives. The batch-mode heuristics were able to execute 50,000 mapping events because we are using a dropping threshold of 0.5 and this makes the heuristics execute quickly. The performance benefit is due to the heuristics using new information to quickly re-map tasks. However, the increase in performance is small because very few tasks among the newly arrived tasks would be *critical* or *high* priority tasks. It is usually the high utility-earning tasks that change the mapping of the previously mapped tasks. As mentioned in Appendix D, available in the online supplemental material, on average approximately 4 and 11 percent of tasks are *critical* and *high* priority tasks, respectively. Therefore, after a minute or after 35 task arrivals, there would probably be approximately one *critical* and four *high* priority tasks among the newly arrived tasks. Scheduling these as soon as they arrive instead of waiting for less than a minute provides only a marginal increase in the total performance.

8 Conclusions and Future Work

In this study, we develop a flexible metric that uses utility functions to compare the performance of resource allocation heuristics in an oversubscribed heterogeneous computing environment where tasks arrive dynamically throughout a 24 hour period. We model this type of environment based on the needs of the ESSC at ORNL. We design and analyze the performance of seven immediate-mode heuristics and five batch-mode heuristics in our simulated environment based on the total utility they could earn during a one day time period. We observe that without the ability to drop tasks, the naive immediate-mode heuristics perform poorly compared to the smart immediate-mode heuristics, which in turn perform poorly compared to the batch-mode heuristics. Among the batch-mode heuristics, Max-Max UPT and MET-Max Util-Max UPT perform the best. This is because these batch-mode heuristics consider the minimization of the execution time of the task in addition to maximizing utility. This is helpful in an oversubscribed highly heterogeneous environment. Dropping low utility-earning tasks significantly helps improve performance of the immediate-mode heuristics because it allows other relatively high-utility earning tasks to execute sooner and thus earn more utility. Dropping tasks also improves the batch-mode heuristics in two ways, (a) by preventing large batch sizes which results in more mapping events being triggered due to faster heuristic execution times, and (b) by preventing lower-priority tasks from entering into the pending slot so that higher priority tasks that arrive subsequently can execute sooner. Immediate-mode heuristics are much more sensitive to the value of the dropping threshold and rely on its tuning to avoid low utility earning tasks from entering machine queues. Permuting the initial tasks at the head of the virtual queues does not affect the performance significantly in our environment. We also experiment with different triggers for the batch-mode mapping events. We observe that (in our environment) triggering every time a new task arrives is not providing significant benefit in the total utility earned compared to mapping after every minute.

Possible direction for future research include: (a) using stochastic estimates of execution time to more closely model a real environment and to analyze the tolerance of the resource management policies to such uncertainties, (b) studying the impact of varying the levels of heterogeneity on heuristic performance, (c) obtaining a model of expected arrival time of tasks from historical data so that the dropping threshold could be varied dynamically throughout the day based on the expected system load, (d) introducing utility functions that do not have to be monotonically-decreasing, (e) introducing parallel jobs (that require multiple machines concurrently to execute), (f) permitting preemption of tasks, (g) developing heuristics that take the utility-functions' slopes into consideration to guide their resource allocation decisions, and (h) adapting our heuristics to work with a semi-distributed resource manager [34].

Acknowledgments

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, supported by the Extreme Scale Systems Center at ORNL, which is supported by the Department of Defense. Additional support was provided by a National Science Foundation Graduate Research Fellowship, and by NSF Grant CCF-1302693. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research also used the CSU ISTeC Cray System supported by NSF Grant CNS-0923386. A preliminary version of portions of this material appeared in [35]. This work builds upon the workshop paper with the design and analysis of: a technique to drop low-utility earning tasks, mapping events that trigger only after previous one finishes execution, new and modified heuristics, machine types grouping together machines with similar performance capabilities, more realistic task arrival patterns and ETC models based on the needs of the ESSC, batches of different sizes, operation of permuting, etc. The authors thank S. Pasricha, S. Powers, G. Pfister, J. Potter, and T. Hansen for their valuable comments.

References

[1] E. Jensen, C. Locke, and H. Tokuda, "A time-driven scheduling model for real-time systems," in *Proc. Int. Real-Time Syst. Symp.*, Dec. 1985, pp. 112–122.

[2] B. Ravindran, E. D. Jensen, and P. Li, "On recent advances in time/utility function real-time scheduling and resource management," in *Proc. Int. Symp. Object-Oriented Real-Time Distrib. Comput.*, May 2005, pp. 55–60.

[3] K. Chen and P. Muhlethaler, "A scheduling algorithm for tasks described by time value function," *J. Real-Time Syst.*, vol. 10, no. 3, pp. 293–312, May 1996.

[4] M. Kargahi and A. Movaghar, "Performance optimization based on analytical modeling in a real-time system with constrained time/utility functions," *IEEE Trans. Comput.*, vol. 60, no. 8, pp. 1169–1181, Aug. 2011.

[5] C. B. Lee and A. E. Snavely, "Precise and realistic utility functions for user-centric performance analysis of schedulers," in *Proc. Int. Symp. High Perform. Distrib. Comput.*, 2007, pp. 107–116.

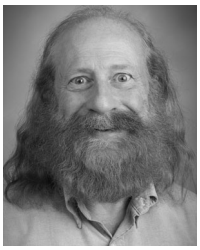
[6] J.-K. Kim, H. J. Siegel, A. A. Maciejewski, and R. Eigenmann, "Dynamic resource management in energy constrained heterogeneous computing systems using voltage scaling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 11, pp. 1445–1457, Nov. 2008.

[7] 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," *J. Parallel Distrib. Comput.*, vol. 59, no. 2, pp. 107–121, Nov. 1999.
[8] B. D. Young, J. Apodaca, L. D. Briceo, J. Smith, S. Pasricha, A. A. Maciejewski, H. J. Siegel, B. Khemka, S. Bahirat, A. Ramirez, and Y. Zou, "Deadline and energy constrained dynamic resource allocation in a heterogeneous computing environments," *J. Supercomput.*, vol. 63, no. 2, pp. 326–347, Feb. 2013.
[9] H. Barada, S. M. Sait, and N. Baig, "Task matching and scheduling in heterogeneous systems using simulated evolution," in *Proc. Int. Heterogeneity Comput. Workshop*, Apr. 2001, pp. 875–882.
[10] M. K. Dhodhi, I. Ahmad, and A. Yatama, "An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems," *J. Parallel Distrib. Comput.*, vol. 62, no. 9, pp. 1338–1361, Sep. 2002.
[11] A. Ghafoor and J. Yang, "A distributed heterogeneous supercomputing management system," *IEEE Comput.*, vol. 26, no. 6, pp. 78–86, Jun. 1993.
[12] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *IEEE Concurrency*, vol. 6, no. 3, pp. 42–51, Jul. 1998.
[13] A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C. Wang, "Heterogeneous computing: Challenges and opportunities," *IEEE Comput.*, vol. 26, no. 6, pp. 18–27, Jun. 1993.
[14] D. Xu, K. Nahrstedt, and D. Wichadakul, "QoS and contention-aware multi-resource reservation," *Cluster Comput.*, vol. 4, no. 2, pp. 95–107, Apr. 2001.
[15] M. R. Gary and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA, USA: Freeman, 1979.
[16] C. M. Krishna and K. G. Shin, *Real-Time Systems*. New York, NY, USA: McGraw-Hill, 1997.
[17] 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," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, Jun. 2001.
[18] L. D. Briceño, H. J. Siegel, A. A. Maciejewski, M. Oltikar, J. Brateman, J. White, J. Martin, and K. Knapp, "Heuristics for robust resource allocation of satellite weather data processing onto a heterogeneous parallel system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 11, pp. 1780–1787, Nov. 2011.
[19] V. Shestak, J. Smith, H. J. Siegel, and A. A. Maciejewski, "Stochastic robustness metric and its use for static resource allocations," *J. Parallel Distrib. Comput.*, vol. 68, no. 8, pp. 1157–1173, Aug. 2008.
[20] P. Sugavanam, H. J. Siegel, A. A. Maciejewski, M. Oltikar, A. Mehta, R. Pichel, A. Horiuchi, V. Shestak, M. Al-Otaibi, Y. Krishnamurthy, S. Ali, J. Zhang, M. Aydin, P. Lee, K. Guru, M. Raskey, and A. Pippin, "Robust static allocation of resources for independent tasks under makespan and dollar cost constraints," *J. Parallel Distrib. Comput.*, vol. 67, no. 4, pp. 400–416, Apr. 2007.
[21] I. Al-Azzoni and D. G. Down, "Linear programming-based affinity scheduling of independent tasks on heterogeneous computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 12, pp. 1671–1682, Dec. 2008.
[22] J.-K. Kim, S. Shivle, H. J. Siegel, A. A. Maciejewski, T. Braun, M. Schneider, S. Tideman, R. Chitta, R. B. Dilmaghani, R. Joshi, A. Kaul, A. Sharma, S. Sripada, P. Vangari, and S. S. Yellampalli, "Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment," *J. Parallel Distrib. Comput.*, vol. 67, no. 2, pp. 154–169, Feb. 2007.
[23] S. Ghanbari and M. R. Meybodi, "On-line mapping algorithms in highly heterogeneous computational grids: A learning automata approach," in *Proc. Int. Conf. Inf. Knowl. Technol.*, May 2005.
[24] Q. Ding, and G. Chen, "A benefit function mapping heuristic for a class of meta-tasks in grid environments," in *Proc. Int. Symp. Cluster Comput. Grid*, May 2001, pp. 654–659.
[25] K. Kaya, B. Ucar, and C. Aykanat, "Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories," *J. Parallel Distrib. Comput.*, vol. 67, no. 3, pp. 271–285, Mar. 2007.
[26] S. Shivle, H. J. Siegel, A. A. Maciejewski, P. Sugavanam, T. Banka, R. Castain, K. Chindam, S. Dussinger, P. Pichumani, P. Satyasekaran, W. Saylor, D. Sendek, J. Sousa, J. Sridharan, and J. Velazco, "Static allocation of resources to communicating subtasks in a heterogeneous ad hoc grid environment," *J. Parallel Distrib. Comput.*, vol. 66, no. 4, pp. 600–611, Apr. 2006.
[27] M. Wu and W. Shu, "Segmented min-min: A static mapping algorithm for meta-tasks on heterogeneous computing systems," in *Proc. Int. Heterogeneity Comput. Workshop*, Mar. 2000, pp. 375–385.
[28] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, S. Spring, A. Su, and D. Zagorodnov, "Adaptive computing on the grid using AppLeS," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 4, pp. 369–382, Apr. 2003.
[29] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for scheduling parameter sweep applications in grid environments," in *Proc. Int. Heterogeneity Comput. Workshop*, Mar. 2000, pp. 349–363.
[30] M. Snir and D. A. Bader, "A framework for measuring supercomputer productivity," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 4, pp. 417–432, Nov. 2004.
[31] P. Li, B. Ravindran, H. Cho, and E. D. Jensen, "Scheduling distributable real-time threads in Tempus middleware," in *Proc. Int. Conf. Parallel Distrib. Syst.*, 2004, pp. 187–194.
[32] 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 Trans. Softw. Eng.*, vol. 30, no. 9, pp. 613–629, Sep. 2004.
[33] S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, and S. Ali, "Representing task and machine heterogeneities for heterogeneous computing systems," *Tamkang J. Sci. Eng.*, vol. 3, no. 3, pp. 195–207, Nov. 2000.
[34] I. Ahmad and A. Ghafoor, "Semi-distributed load balancing for massively parallel multicomputer systems," *IEEE Trans. Softw. Eng.*, vol. 17, no. 10, pp. 987–1004, Oct. 1991.
[35] L. D. Briceño, B. Khemka, H. J. Siegel, A. A. Maciejewski, C. Groer, G. Koenig, G. Okonski, and S. Poole, "Time utility functions for modeling and evaluating resource allocations in a heterogeneous computing systems," in *Proc. 20th Heterogeneity Comput. Workshop*, May 2011, pp. 7–19.

Bhavesh Khemka received the BE degree in electrical and electronics engineering from the Hindustan College of Engineering affiliated with Anna University, India. He is currently working toward the PhD degree and is a research assistant in the Department of Electrical and Computer Engineering at Colorado State University. His research interests include fault-tolerant, energy-aware, and robust resource allocations in heterogeneous and distributed computing environments.

Ryan Friese received dual BS degrees in computer engineering and computer science from Colorado State University in 2011. He received the MS degree in electrical engineering from Colorado State University in the Summer of 2012. He is currently working toward the PhD degree at Colorado State University. He is a United States National Science Foundation graduate research fellow. His research interests include heterogeneous computing as well as energy-aware resource allocation.

Luis D. Briceño received the BS degree in electrical and electronic engineering from the University of Costa Rica, and the PhD degree in electrical and computer engineering at Colorado State University. His research interests include heterogeneous parallel and distributed computing.



Howard Jay Siegel received the BS degrees from MIT, and the PhD degree from Princeton. He was appointed the Abell Endowed Chair Distinguished professor of electrical and computer engineering at Colorado State University (CSU) in 2001, where he is also a professor of computer science. From 1976 to 2001, he was a professor at Purdue. He is a fellow of the IEEE and ACM.

Anthony A. Maciejewski received the BSEE, MS, and PhD degrees from The Ohio State University in 1982, 1984, and 1987, respectively. From 1988 to 2001, he was a professor of Electrical and Computer Engineering at Purdue University, West Lafayette. He is currently a professor and department head of electrical and computer engineering at Colorado State University. He is a fellow of the IEEE.

Gregory A. Koenig received three BS degrees in mathematics, electrical engineering technology, and computer science from Indiana University-Purdue University Fort Wayne in 1996, 1995, and 1993, respectively. He received the MS and PhD degrees in computer science from the University of Illinois at Urbana-Champaign in 2003 and 2007, respectively. He is an R&D staff member at Oak Ridge National Laboratory, where his work involves developing scalable system software and tools for ultrascale-class parallel computers.

Chris Groer received the PhD degree from the University of Maryland. He worked as a research mathematician for the Department of Defense and Oak Ridge National Lab. He leads Research and Development at Link Analytics, a company focused on applying mathematics and operations research to problems in telecommunications, networking, big data analytics, and scheduling. He is also a former professional tennis player and was inducted into the Vanderbilt Hall of Fame in 2010.

Gene Okonski received the BS degree in electrical engineering with a minor in economics from Colorado State University in 1988. He received the MS degree in engineering from the Johns Hopkins University School of Engineering with a concentration in systems engineering in 1993. He is a systems architect for the Department of Defense. His areas of focus are on distributed computing, data processing architectures, and information technology efficiency. He has been involved as a technologist and leader of a number of large scale development activities spanning some 20 years.

Marcia M. Hilton received the bachelor of science degree in computer science from the University of Kentucky in 1997. She currently works for the Department of Defense specializing in Application Frameworks and Automatic Processing Systems. She is a project technical lead on an Automatic Data Processing System that makes use of commercial schedulers to execute a variety of well defined tasks. She is involved in the implementation of a stable meta-scheduler that will enable a diverse set of customers to intelligently share a common set of resources.

Rajendra Rambharos received the BS degree in computer engineering from the University of Central Florida in 2006. After graduation, he began work with Harris Corporation, headquartered in Melbourne, Florida, where he gained experience working on the software systems for the FAA telecommunications network for air traffic control. Following Harris, he joined the Department of Defense, working on automated data processing systems in a high availability environment. He is currently a software and systems developer for an intelligent task scheduler architecture.

Steve Poole currently divides his time between duties at the Department of Defense and Oak Ridge National Laboratory. He is the chief scientist and director of Special Programs. He can be reached at spoole@ornl.gov.