Stochastic-Based Robust Dynamic Resource Allocation for Independent Tasks in Heterogeneous Computing System

Mohsen Amini Salehi[a,*], Jay Smith[b], Anthony A. Maciejewski[c], Howard Jay Siegel[c], Edwin K. P. Chong[c], Jonathan Apodaca[c], Luis D. Briceño[d], Timothy Renner[e], Vladimir Shestak[b], Joshua Ladd[f], Andrew Sutton[e], David Janovy[g], Sudha Govindasamy[d], Amin Alqudah[h], Rinku Dewri[i], Puneet Prakash[j]

[a]*High Performance Cloud Computing (HPCC) Laboratory, School of Computing and Informatics, University of Louisiana, Lafayette, LA 70503, USA*
[b]*Lagrange Systems, Boulder, CO 80302, USA*
[c]*Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO 80523, USA*
[d]*Intel Inc.*
[e]*Department of Computer Science, Colorado State University, Fort Collins, CO 80523, USA*
[f]*Mellanox Technologies Inc.*
[g]*BHGrid Inc.*
[h]*Department of Computer Engineering, Yarmouk University, Jordan*
[i]*Department of Computer Science, University of Denver*
[j]*Environmental System Research Institute (Esri)*

[*]Corresponding author. Telephone: +1-337-482 5807; Fax: +1-337-482 5791
Email addresses: amini@louisiana.edu (Mohsen Amini Salehi), jay@lagrangesystems.com (Jay Smith), aam@colostate.edu (Anthony A. Maciejewski), hj@colostate.edu (Howard Jay Siegel), edwin.chong@colostate.edu (Edwin K. P. Chong), jonathan.apodaca@colostate.edu (Jonathan Apodaca), luis.d.briceno.guerrero@intel.com (Luis D. Briceño), timothy.renner@gmail.com (Timothy Renner), vladimir@lagrangesystems.com (Vladimir Shestak), joshual@mellanox.com (Joshua Ladd), sutton@cs.colostate.edu (Andrew Sutton), djanovy@bhgrid.com (David Janovy), sudha.govindasamy@intel.com (Sudha Govindasamy), amin.alqudah@yu.edu.jo (Amin Alqudah), rdewri@cs.du.edu (Rinku Dewri), puneet_prakash@esri.com (Puneet Prakash)

Preprint submitted to Journal of Parallel and Distributed Computing *May 9, 2016*

Abstract

Heterogeneous parallel and distributed computing systems frequently must operate in environments where there is uncertainty in system parameters. Robustness can be defined as the degree to which a system can function correctly in the presence of parameter values different from those assumed. In such an environment, the execution time of any given task may fluctuate substantially due to factors such as the content of data to be processed. Determining a resource allocation that is robust against this uncertainty is an important area of research. In this study, we define a stochastic robustness measure to facilitate resource allocation decisions in a dynamic environment where tasks are subject to individual hard deadlines and each task requires some input data to start execution. In this environment, the tasks that cannot meet their deadlines are dropped (i.e., discarded). We define methods to determine the stochastic completion times of tasks in the presence of the task dropping. The stochastic task completion time is used in the definition of the stochastic robustness measure. Based on this stochastic robustness measure, we design novel resource allocation techniques that work in immediate and batch modes, with the goal of maximizing the number of tasks that meet their individual deadlines. We compare the performance of our technique against several well-known approaches taken from the literature and adapted to our environment. Simulation results of this study demonstrate the suitability of our new technique in a dynamic heterogeneous computing system.

Keywords: Dynamic resource allocation, heterogeneous computing, robustness, scheduling, stochastic models.

---

1. Introduction

Heterogeneous parallel and distributed computing systems frequently operate in environments where uncertainty in task execution time is common. For instance, the execution time of a task can depend on the data to be processed. We represent each task's execution time on each machine as a probability mass function (pmf). Robustness can be defined as the degree to which a system can maintain a given level of performance even with this uncertainty [1, 2, 3].

One challenge to make a robust system is how to measure and quantify robustness in the system. To address this challenge, one contribution of this research is to design a dynamic stochastic robustness measure for heterogeneous computing (HC) systems. In particular, we investigate a robustness measure for an HC system that evaluates a dynamic (on-line) resource allocation.

A *mapping event* is defined as the time when the resource allocation procedure is executed to map (i.e., assign and schedule) tasks to machines. After a task is mapped to a machine, its required input data is staged (i.e., loaded) to the corresponding machine and then the task can start execution.

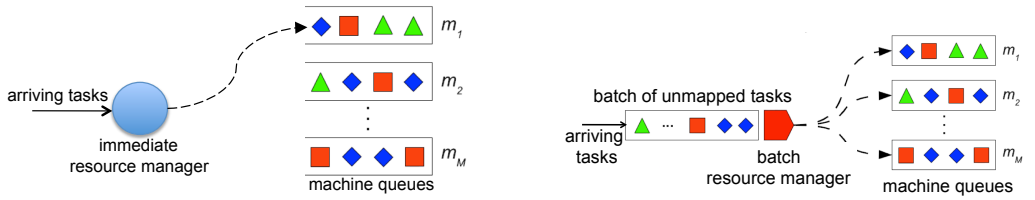
A dynamic resource manager can operate either in *immediate* or *batch* mode [4]. The difference between these resource management approaches is in the way they map arriving tasks to machines. In the immediate mode, shown in Figure 1(a), each task is mapped to one of M machines immediately upon its arrival. In contrast, in one variation of the batch mode, shown in Figure 1(b), a limited number of tasks are mapped to each machine and the rest of them are saved at the resource manager for assignment during the next mapping event, along with newly arriving tasks. These tasks saved at the resource manager and the newly arrived tasks form the set of unmapped tasks. We investigate the performance of both immediate and batch operation modes on HC systems.

In this study, each task has an individual hard deadline. We consider an HC suite of machines that is *oversubscribed*. By oversubscribed we mean that the arrival rate of tasks, in general, is such that the system is not able to complete all tasks by their individual deadlines. Therefore, the research problem we investigate in this work is: How to maximize the number of tasks that are completed by their individual deadlines in an oversubscribed HC system? Accordingly, the performance measure that we consider is the number of tasks that are completed by their individual deadlines.

In this system, because there is no value in executing a task after its deadline, the task is dropped (i.e., discarded) if it misses its deadline. Dropping can also take place as a result of task failure [5]. Dropping a task affects the completion time of the tasks queued behind the dropped task. Hence, we provide a method to determine the stochastic completion time of the tasks in the presence of the dropping. Then, we use the stochastic task completion time to provide a mathematical model to measure the robustness of a resource allocation.

Dropping the tasks in batch mode, where the number of tasks that are mapped to each machine is limited, can potentially lead to the state where there is no task in a machine queue, thus wasting the computational capacity of the machine. To avoid this state, we schedule mapping events to occur before a machine becomes idle. Additionally, the limit on the number of tasks that are mapped to each machine (i.e., machine queue-size limit) is influential on the performance of the batch mode resource allocation. Therefore, another contribution of this study is to verify the proper queue-size limit for a batch mode resource allocation.

In general, the problem of resource allocation in the field of heterogeneous



(a) immediate (b) batch

Figure 1: Dynamic resource allocation that (a) maps arriving tasks to machines immediately upon arrival, and (b) maps a limited number of tasks to each machine and queues the rest of arriving tasks at the resource manager.

parallel and distributed computing is NP-complete (e.g., [6, 7]); hence, the development of heuristic techniques to find near-optimal solutions represents a large body of research (e.g., [8, 9, 10, 11, 12, 13, 14, 15]). Therefore, based on the analysis of the stochastic robustness measure, we design resource allocation heuristics that are capable of allocating a dynamically arriving set of tasks to a dedicated HC system. We compare our robustness-based resource allocation approach against several resource allocation techniques taken from the literature and adapted to this environment. We compare the performance of the mapping heuristics via simulation which allows to evaluate a variety of working conditions. The results of our simulation study demonstrate the efficacy of our robustness-based approach.

We are interested in resource allocation techniques that can tolerate higher levels of over-subscription. Thus, as a contribution of this study, we analyze how different resource allocation techniques perform when the over-subscription level increases in the HC system. Additionally, an ideal resource allocation technique should perform well when tasks have data requirements to start their execution. Hence, another contribution of this study is to analyze the behavior of different resource allocation techniques when tasks require input data.

In summary, this study makes the following contributions:

- Determining the stochastic task completion time in a system where tasks are dropped if they miss their deadlines.
- Using stochastic task completion time to provide a mathematical model for quantifying the robustness of a resource allocation.
- Designing and analyzing novel resource allocation techniques that operate based on our proposed robustness measure.

- Planning mapping events in the batch mode in a way that the computational capacity of machines is not wasted.
- Investigating the performance impact of various queue-size limits for different batch mode resource allocation techniques.
- Analyzing the impact of the over-subscription levels on the performance of different resource allocation techniques.
- Analyzing the behavior of various resource allocation techniques when tasks have data dependencies.

In the next section, we present the system model. A review of the related work is given in Section 3. Section 4 describes our mathematical model of robustness in a dynamic environment. Section 5 examines how machine idling can be avoided in the batch mode resource allocation approach. The heuristic techniques for this environment are given in Section 6. The details of the simulation setup used to evaluate our heuristics are discussed in Section 7. Section 8 provides the results of our simulation study and Section 9 concludes the paper.

2. System Model and Problem Statement

This research was motivated by an *inconsistent heterogeneous* distributed computing system used for image processing [3], similar to those deployed at DigitalGlobe [16]. An inconsistent HC system includes a mixture of different machines to execute tasks with various computational needs. In particular, in such a system, each task may have different execution times on different machines of the system. For instance, machine A may be faster than machine B for task 1 but slower than other machines for task 2. This is because each tasks execution time depends on how the tasks computational needs interact with the machine's capabilities. It is worth noting that this is a general model and includes HC systems with different types of machines, such as those that consist of CPUs and GPUs [17]. Although the methods discussed in this research are designed for inconsistent HC systems, they also work in distributed computing systems where only a portion of machines are heterogeneous.

In this system, user tasks for processing are sent to a resource manager for assignment to any one of a collection of dedicated machines. Each task is compute-intensive and consists of an operation to be executed (e.g., compression, decompression, rotation) plus an input file to be processed. The list of available image

processing operations, from which the user can select, is referred to here as *task types*. It is limited to a set of frequently requested algorithms, such as those found in a research lab or military environments (e.g., [18, 3, 17]). A problem arises in trying to complete each task by its individual deadline because the HC suite is oversubscribed and there are uncertainties in the tasks' execution times.

In our system model, tasks arrive dynamically, and the exact sequence of the arrivals is not known in advance. Each arriving task request r_i requires some input data that has to be loaded from a shared storage for execution. Additionally, each task r_i is assigned an individual hard deadline for completion, denoted δ_i. By *hard deadline*, we mean that if a task cannot be completed before its deadline, it is dropped.

Dropping tasks is a common practice in oversubscribed systems that have a real time or near real time nature (e.g., [19, 20, 21]). In these systems, typically, there is no value in executing a task that has missed its deadline. For instance, in live video streaming [22] systems, each frame should be transformed (i.e., transcoded) based on the client's machine characteristics. The transcoding operation of each frame must be completed within a tight deadline. However, there is no value in transcoding frames that have missed their deadlines. Dropping these frames help other frames queued in the system to be transcoded before their deadlines. Another instance of task dropping is in systems that process periodically received data sets (e.g., security surveillance [23] and medical images processing [24]). Usually, in these systems, sensors periodically produce data sets. The processing of each data set must be completed before arrival of the next data set. That is, there is no value in processing a data set after arrival of the next data set. In an oversubscribed system, dropping tasks that miss their deadlines reduces the waiting times and increases the likelihood of meeting deadline for other queued tasks. In addition to these motivations, dropping is usually unavoidable when a task failure occurs in a system [5].

Task dropping can occur either at mapping time or before starting execution on a machine. Also, a currently executing task can be dropped as soon as it misses its deadline. However, dropping tasks while they are executing is not possible in some systems. For instance, in [3], a stream of images has to be processed for rasterization and displaying within a tight deadline. In this system, once the processing of an image is started, it has to be completed. Dropping of the executing tasks is also not possible for database transactional tasks (where transactions have to complete their executions, once started, to maintain the consistency of the data [25]) and in real time systems [26]. Therefore, in this study, we consider two scenarios: in the *first* scenario the currently executing task is dropped as soon

as it misses its deadline and in the *second* scenario the executing task cannot be dropped and must complete its execution.

The exact execution time of any given task on a given machine is assumed to be dependent on the characteristics of the data that is to be processed (including the size and actual content of the data). Therefore, the execution time for a given task can be highly variable and, as such, is treated as a random variable. We assume that a discrete probability distribution, known as a probability mass function (*pmf*), is available for each task type's execution time on each machine. That is, each task type is associated with a set of pmfs, one pmf for each machine in the HC suite, describing the probability of possible execution times for that task type. A typical method for creating such distributions relies on a histogram estimator [27] that produces pmfs based on historical and analytical techniques [28]. We assume that the collection of task execution time pmfs has been provided in advance.

The uncertainties in the tasks' execution times cause resource allocation decisions to be more difficult. A robust resource allocation technique takes into account the uncertainties in the tasks' execution times [13]. Any claim of robustness for a given system must answer three fundamental questions [2]:

(a) What behavior makes the system robust? A robust resource allocation in this environment is one that is capable of completing tasks by their assigned individual deadlines.

(b) What are the uncertainties that the system is robust against? The execution time for each task on each machine is a known source of uncertainty and is represented by a random variable (pmf).

(c) How is system robustness quantified? The robustness of a resource allocation can be quantified as the expected number of tasks that will complete before their individual deadlines, as predicted at a given point in time.

From this set of requirements, we formulate a robustness measure for a resource allocation in the system.

Uncertainty in task execution time can impact the completion times of all tasks that share the same machine for execution [29]. For example, given multiple tasks allocated to the same machine, a longer than expected execution time for a task early in the queue may cause tasks later in the queue to miss their deadlines. This effect is compounded when multiple tasks take longer than expected. To mitigate the impacts of task execution time uncertainty in batch mode, we chose to limit

the number of tasks that can be queued, denoted L, at any single machine (i.e., machine queue-size limit). Due to the influence of the queue-size limit on the performance of batch mode heuristics, in this research, we determine the proper machine queue-size for the batch mode heuristics.

Another aspect of limiting machine queue-size is the state where no task remains in the machine queue and the machine becomes idle. This is particularly important in our system due to task dropping. The idle machine should wait until the resource allocation heuristic is executed and the input data for a mapped task is staged to the selected machine. We refer to these waiting times as the *mapping overhead*. To avoid machine idling, in this research, we perform mapping events before the machines become idle.

In batch mode resource allocation, the remaining tasks that are not already in a machine queue as well as tasks that have arrived since the last mapping event form a batch of tasks at the resource manager (see Figure 1(b)). Hence, at each time-step[1] $t^{(k)}$, we effectively define a batch of tasks at the resource manager, denoted $B(k)$. Based on this definition, a mapping event in the batch mode resource allocation occurs due to one of two conditions:

(a) There is at least one task in $B(k)$ and at least one free space is created in the machine queues (e.g., due to completion of the currently executing tasks).

(b) There is a free space in the machine queues and a new task arrives to the batch queue ($B(k)$).

In contrast, in the immediate mode resource allocation a mapping event occurs upon arrival of a new task. There is no limit on machines' queue-sizes in this case.

For both immediate and batch modes, after a task is mapped, it is placed in the input queue of its allocated machine (see Figure 1) and input data for the task is staged to the machine. Due to the heterogeneous nature of our system, we consider a different data staging rate to each machine in the HC system. For this study, due to the overhead of data transfer, once a task has been queued for execution on a machine, it cannot be re-allocated to any other machine.

We assume that the HC suite operates in a non-multi-tasking mode; i.e., each machine only executes one task at a time, as is the case with the cores in the ISTeC Cray XT6m system currently in use at Colorado State University [30]. Addition-

[1]In this research, the time of the system is modeled in the form of discrete steps, each one called a time-step.

ally, each task's execution time (not completion time) is assumed to be independent, i.e., there is no inter-task communication. This assumption of independence is valid for non-multitasking execution mode, which is commonly considered in the literature (e.g., [31, 32, 33]).

3. Related Work

The problem of workload distribution considered in our research falls into the category of dynamic resource allocation. The general problem of dynamically allocating independent tasks to HC systems was studied in [4]. The primary objective in [4] was to minimize system makespan, i.e., the total time required to complete a set of tasks. This objective is different from the primary objective in our current work: to complete each task before its deadline. Furthermore, in [4] the task execution times are assumed to be deterministic not stochastic.

Paragon [34] is a scalable immediate mode resource allocation method for large-scale heterogeneous Cloud datacenters. It uses historic information of the tasks' execution times to classify an unknown arriving task with respect to the machine heterogeneity and interference with other co-located tasks. The Paragon's classification engine discovers similarities in the tasks' resource requirements. It uses singular value decomposition to identify similarities between incoming and previously scheduled tasks. Once an incoming task is classified, a greedy mapping heuristic assigns it to a machine with the goals of minimizing the task's completion time and maximizing the machine utilization. The heuristic searches for machines whose current load can tolerate the interference caused by the new task. Then, from the set of candidate machines, it selects the machine that provides the minimum execution time for the arriving task. This work is different from our research in that it focuses on the scalability of the resource allocation for large-scale datacenters and utilizes a scalar execution time to estimate the fitness of a machine for a given task. Furthermore, the tasks do not have deadlines, and the performance goal is different.

Mapping heuristics are proposed in [35] for the immediate mode resource allocation in an HC system. The proposed mapping heuristics utilize the arrival rate information of different task types to decrease waiting times of the tasks and guarantee the stability of the HC system. However, stochasticity in the tasks' execution times is not considered in [35] and further studies are required to examine the stability of the HC system when such uncertainties exist in the system. In addition, the proposed heuristics in [35] have the knowledge of each task type arrival rate distribution whereas our proposed heuristics do not have any assumption

about the arrival rates distributions.

In [36], a stochastic task mapping technique is provided for HC systems where uncertainty exists in the tasks' execution times. The stochastic task mapping is formulated as a linear programming problem with the goal of creating a balance between the makespan of each Bag of Tasks (BoT) application with a collective deadline and the energy consumption of the application. The proposed mapping technique can improve the weighted probability that both the deadline and the energy consumption budget constraints can be met. In contrast, our research goal is to increase the probability of meeting the individual deadlines of dynamically arriving tasks. Also, we consider the impact of dropping tasks on the stochastic completion time of other tasks in the system.

Xu et al. [37] propose a resource allocation method for HC systems with the goal of minimizing the makespan of a workflow. Their idea is to incorporate a Genetic Algorithm based technique to assign priority to each task of the workflow while using the earliest completion time heuristic to map the tasks to machines. The Genetic Algorithm based technique prioritizes the tasks in a way that the makespan of the workflow is minimized. Then, the mapping heuristic selects the tasks based on their assigned priorities and maps each of them to the machine that offers the minimum completion time for that task. The authors assume that the exact execution time of the tasks are known apriori whereas we consider stochastic task execution times on each machine. Also, the goal of the resource allocation is different.

Canon et al. [1] investigate static scheduling techniques to maximize the robustness and minimize the makespan for workflow applications in an HC environment. They define the robustness as the stability of the makespan for any realization of the same schedule for a given workflow. The stability of the makespan is modeled by the inverse standard deviation of the makespan distribution. Because the robustness and the makespan are not equivalent objectives, they apply a bi-criteria approach to find all the Pareto-optimal solutions and investigate the trade-off between the two metrics. In contrast, we consider the problem of dynamic resource allocation for independent tasks with the goal of maximizing the number of tasks that meet their deadlines.

In [13], a stochastic robustness measure is developed for static resource allocation in a heterogeneous distributed system, where all the task arrivals are known before the heuristics begin. In this method, the mapping heuristic tries to keep the robustness measure between predefined boundaries. The authors of the paper have studied an iterative mapping heuristic that starts with a complete allocation and progressively modifies the order and evaluates it until a desired performance is

reached. Our work is different from [13] from several aspects. First and the foremost is the fact that we investigate dynamic (on-line) resource allocation whereas in [13] a static resource allocation technique is studied. The allocation heuristic proposed in [13] is an off-line method and cannot be utilized for dynamic (on-line) systems. The second difference is that the stochastic robustness measure in [13] is different from what we consider here. The third difference is that we investigate and provide a model for stochastic completion time of tasks in the presence of task dropping.

4. Mathematical Model

4.1. Stochastic Task Completion Time

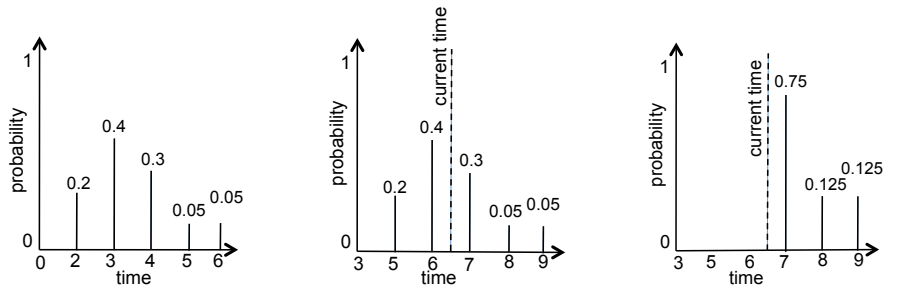
We define new methods for determining the completion time of a given task r_i at time-step $t^{(k)}$. This builds on our previous method in [38]. More specifically, we introduce methods to determine the task completion time for three different cases. The first case (explained in Subsection 4.1.1), is when there is no task dropping. This method was initially introduced in [38]. However, we describe it here because it serves as a basis for other two unexplored cases where task dropping is allowed. The second case (explained in Subsection 4.1.2), determines the task completion time for the scenario where a pending task (i.e., a task waiting for execution in a machine queue) is dropped if the current time exceeds its deadline. However, in this case, a currently executing task cannot be dropped and has to complete its execution even if it misses its deadline. The third case (explained in Subsection 4.1.3), provides a method for the task completion time when any task (either pending or executing task) is dropped if the current time exceeds its deadline.

Let $\mu^{(k)}$ denote the set of all tasks that are either queued for execution or currently executing on any of the M machines in the HC suite at time-step $t^{(k)}$. Let the ordered list of tasks that were assigned to machine j in advance of task r_i but that have not yet completed execution as of $t^{(k)}$ be denoted $\mu_{ij}^{(k)}$.

4.1.1. No Task Dropping

Because the execution time of each task in the system is a random variable, the distribution of the completion time of task r_i is found as the convolution [39] of the execution time probability distributions for all tasks either currently executing or pending execution in advance of task r_i on the same machine.

To find the completion time pmf for a currently executing task z on a given machine j, we must account for impulses (also known as atoms) in the execution



(a) execution time distribution

(b) completion time distribution for start time of 3

(c) renormalized completion time distribution at time 6.5

Figure 2: Execution and completion time distributions for a given task that is being executed in a machine. (a) Execution time distribution of the task. (b) Given that the task started execution at time 3, the completion time distribution for the task after shifting the task execution time distribution. (c) Completion time distribution after renormalizing based on the current time of 6.5 by removing impulses that occur before the current time.

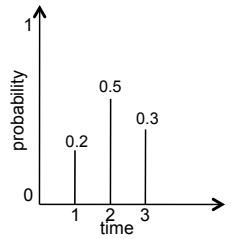
time pmf of task z that would have occurred prior to the current time-step $t^{(k)}$. For example, if task z began execution at time-step $t^{(h)}$ $(h < k)$, then we know that all impulses in the completion time distribution for task z with values less than $t^{(k)}$ did not occur. Thus, accurately describing the completion time of task z at time-step $t^{(k)}$ requires that these past impulses be removed from the pmf and the remaining distribution be renormalized to 1.

Figure 2 describes how renormalization of a completion time pmf is performed for a given task that is currently being executed in a machine. Figure 2(a) shows the execution time pmf for the task. Figure 2(b) shows the completion time pmf for the task that is constructed by shifting the execution time pmf based on the task's start time (which is 3 in this example). Figure 2(c) shows that for renormalization, past impulses (i.e., impulses before the current time) are removed and the remaining impulses are renormalized and form the completion time pmf for the task at time 6.5.

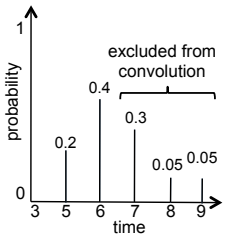
We can find the completion time distribution of task r_i at time-step $t^{(k)}$ by convolving the completion time distribution of the currently executing task on machine j with the execution time distributions of all pending tasks in $\mu_{ij}^{(k)}$. Finally, the resulting completion time pmf is convolved with the execution time distribution for task r_i on machine j.

4.1.2. Dropping Pending Tasks

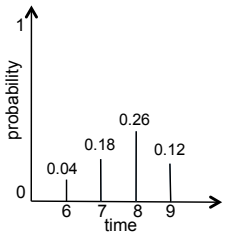
Dropping a pending task in $\mu_{ij}^{(k)}$ influences the completion time of the tasks queued behind the dropped task on machine j. However, the method provided in the previous subsection does not consider the influence of dropping a pending task on the completion time of the tasks queued behind that.



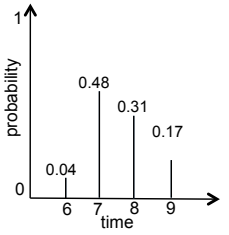
(a) execution time distribution of task r_i with $\delta_i = 7$



(b) completion time distribution of task r_{i-1} with $\delta_{i-1} = 9$



(c) result of convolution between the distribution of task r_i with impulses less than 7 of task r_{i-1}



(d) completion time distribution for task r_i

Figure 3: Completion time distribution for a given task r_i in a system where pending tasks are dropped if they miss their deadlines. (a) Execution time distribution of task r_i with $\delta_i = 7$. (b) Completion time distribution of task r_{i-1} with $\delta_{i-1} = 9$. (c) Distribution resulted from convolving impulses of task r_i with impulses less than time 7 of task r_{i-1}. (d) Completion time distribution for task r_i; obtained from adding impulses greater than or equal to 7 in completion time pmf of task r_{i-1} in Figure 3(b), for which task r_i is dropped and has execution time zero, with the distribution in Figure 3(c).

Intuitively, pending task r_i is dropped if task r_{i-1} is completed at or after the deadline of task r_i. That is, the probability of dropping for task r_i is the sum of probabilities of impulses in the completion time pmf of task r_{i-1} that are greater than or equal to δ_i. Therefore, in the first step to obtain the completion time pmf

for task r_i, just the impulses in the completion time pmf of task r_{i-1} that are less than δ_i are considered for convolution with the execution time pmf of task r_i.

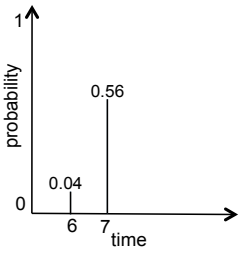
Dropping a pending task is equivalent to considering zero execution time for that task. More specifically, because of dropping, for all impulses greater than or equal to δ_i in completion time pmf of r_{i-1} (i.e., impulses that were excluded from convolution in the first step), the execution time pmf of r_i is an impulse at time zero with probability 1. Convolving the excluded impulses with this execution time pmf determines the completion time pmf of task r_i when the excluded impulses in completion time distribution of task r_{i-1} occur and complements the distribution obtained in step 1. It is worth noting that convolving a given impulse ψ in the completion time pmf of task r_{i-1}, that occurs at time $t_\psi \geq \delta_i$ with probability p_ψ, with the impulse at time zero and probability 1, will result into an impulse in the completion time of r_i at time $t_\psi + 0$ with probability $1 \times p_\psi$. This is equivalent to adding excluded impulses in the completion time pmf of task r_{i-1} to the completion time pmf of task r_i. Therefore, in the second step, the impulses in the completion time pmf of task r_{i-1} that were excluded from the convolution in step 1, are added to the obtained distribution and form the completion time pmf of task r_i.

Figure 3 depicts the details of determining completion time pmf of r_i when pending tasks in $\mu_{ij}^{(k)}$ are dropped if they miss their deadlines. Figure 3(a) shows the execution time pmf of a given task r_i with $\delta_i = 7$. Figure 3(b) shows the completion time pmf of task r_{i-1} with $\delta_{i-1} = 9$. Figure 3(c) shows the result of convolution between the impulses that are less than $\delta_i = 7$ in completion time pmf of task r_{i-1} (i.e., impulses at times 5 and 6 in Figure 3(b)) with the execution time pmf of task r_i. Next, we consider the impact of impulses that are excluded from convolution in Figure 3(b) on the completion time pmf of task r_i. As mentioned earlier (see the above paragraph), for these excluded impulses in the completion time distribution of task r_{i-1}, the execution time of task r_i is zero. Therefore, the excluded impulses have to be added to the completion time distribution of task r_i. Thus, in Figure 3(d) the completion time distribution of task r_i is formed by adding impulses that are greater than or equal to $\delta_i = 7$ in the completion time pmf of task r_{i-1} (impulses at times 7, 8, and 9 in Figure 3(b)) to the distribution shown in Figure 3(c).

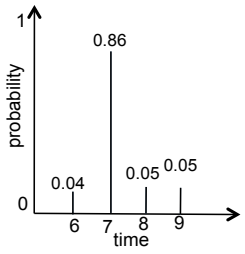
4.1.3. Dropping Pending and Executing Tasks

The method of determining completion time distribution for task r_i, described in the previous subsection, does not consider dropping for a currently executing task. That is, if a task starts executing, it has to be completed even if it misses its

deadline during the execution. In this part, we extend the method provided in the previous subsection to determine the completion time distribution of pending task r_i for the scenario where a currently executing task is dropped as soon as it misses its deadline. In addition, similar to Subsection 4.1.2, a pending task that misses its deadline is also dropped and cannot start execution.



(a) impulses greater than or equal to $\delta_i = 7$ in Figure 3(c) are summed and form an impulse at $\delta_i = 7$



(b) completion time distribution for task r_i

Figure 4: Completion time distribution of task r_i in a system where the currently executing task is dropped as soon as it misses its deadline. (a) After convolving the distribution of task r_i with impulses less than time 7 in the distribution of task r_{i-1} in Figure 3(c), impulses greater than or equal to the deadline of task r_i in the resulting distribution are summed and form an impulse at $\delta_i = 7$. (b) Completion time distribution for task r_i; obtained from adding the distribution in Figure 4(a) with the excluded impulses of task r_{i-1} in Figure 3(b), for which task r_i is dropped and has execution time zero.

To determine the completion time distribution of task r_i that is pending in a machine queue, similar to the previous subsection, in the first step, impulses in the completion time pmf of task r_{i-1} that are less than δ_i are convolved with the execution time pmf of task r_i. Because task r_i will not be executed after its deadline, in the resulting distribution, all the impulses that are greater than δ_i would not occur. Therefore, in the second step, the impulses that are greater than or equal to δ_i in the resulting distribution are summed and form an impulse at δ_i.

Recall from Subsection 4.1.2 that for all the impulses greater than δ_i in the completion time pmf of task r_{i-1}, task r_i is dropped and will have zero execution time. Therefore, in the third step, to consider zero execution time in the completion time distribution of task r_i, similar to Subsection 4.1.2, the impulses in the completion time pmf of task r_{i-1} that were excluded from the convolution are added to the obtained distribution in the second step and form the completion time

pmf of task r_i.

Figure 4 extends the example in Figure 3 to show the details of determining task completion time for the scenario where the executing task is dropped as soon as it misses its deadline. Remember that Figure 3(c) shows the result of convolution between the execution time pmf of task r_i (in Figure 3(a)) with impulses lower than $\delta_i = 7$ in the completion time distribution of task r_{i-1} (in Figure 3(b)). Then, according to the second step, in Figure 4(a), the impulses greater than or equal to $\delta_i = 7$ in Figure 3(c) are summed and form an impulse at the deadline of task r_i. Finally, according to the third step, in Figure 4(b), the completion time distribution of task r_i, is obtained from adding the excluded impulses in the completion time distribution of task r_{i-1} in Figure 3(b) (for which the execution time of task r_i is zero) to the distribution in Figure 4(a).

It is worth noting that the completion time distribution of the currently executing task is obtained by removing past impulses and renormalizing the remaining impulses to 1 (see Subsection 4.1.1). Then, all impulses that are greater than or equal to the task's deadline in the renormalized distribution are summed and form an impulse at the deadline of the task.

4.2. Calculating Stochastic Robustness

The robustness of a resource allocation at time-step $t^{(k)}$ is defined based on the expected number of tasks that meet their individual deadlines, predicted at this time-step. We calculate the total expected number of tasks that meet their individual deadlines by summing over all machines the expected number of tasks on each machine that meet their individual deadlines.

The probability that task r_i completes before its deadline (δ_i) on machine j, denoted $p(r_{ij})$, is calculated by summing the probabilities of impulses that are less than δ_i in the task's completion time pmf. Let n_j the number of tasks assigned to machine j. Then, the stochastic robustness of this machine at time-step $t^{(k)}$, denoted $\rho_j^{(k)}$, is calculated as follows:

$$\rho_j^{(k)} = \sum_{i=1}^{n_j} p(r_{ij}) \tag{1}$$

We define the stochastic robustness of a resource allocation at a given time-step $t^{(k)}$, denoted $\rho^{(k)}$, as the sum of the robustness values associated with each machine. This can be stated formally as follows:

$$\rho^{(k)} = \sum_{\forall j} \rho_j^{(k)} \tag{2}$$

5. Avoiding Machine Idling

Uncertainty in task execution times can affect the completion times of all tasks that share the same machine for execution. For instance, given multiple tasks assigned to the same machine, a longer than expected execution time for a task early in the queue may cause tasks later in the queue to miss their deadlines. This effect is compounded when multiple tasks take longer than expected.

To mitigate the impact of task execution time uncertainty, in batch mode, we chose to limit the number of tasks that can be queued at any single machine. Limiting machine queue-size in an oversubscribed system where tasks are dropped as soon as they miss their deadlines can potentially lead to a circumstance where there is no task in a machine and, therefore, the machine is idle. For instance, machine idling can happen when a currently executing task is completed and the pending tasks are dropped because their deadlines were missed. The idle machine should wait for the mapping overhead. That is, the idle machine should wait until the resource allocation heuristic is executed and the input data for a mapped task is staged to the selected machine. In practice, when the required tasks' input data are substantial, the mapping overhead is not negligible and has to be considered (e.g., [17]). One consequence of ignoring mapping overhead is to delay task execution and to wait for its input data to be staged. Another consequence is to waste the computational capacity of the machine.

One technique often used to avoid machine idling is to pre-stage several tasks into the machine queues in advance of execution (i.e., machine queue-size limit is large). This helps to ensure that there are tasks in the machine queues that are ready for execution and machine idling is avoided. However, a large queue can degrade the performance of a resource allocation heuristic due to the compounded uncertainty in the execution time of tasks that share the same queue. Nonetheless, a short queue-size can potentially lead to machine idling (e.g., when an executing task is completed and pending tasks are dropped from machine queue j).

As demonstrated later (in Section 8.1), to obtain the best performance of a batch mode resource allocation heuristic, we choose to keep the queue-size short. In this case, each machine includes one executing task and one pending task (i.e., the machine queue-size limit is two). However, to avoid machine idling, we propose to perform mapping events before the machines become idle. The time for these mapping events is determined based on the time required for executing the resource allocation heuristic, denoted t_h, and the time associated with staging the tasks' input data to machine j, denoted t_{lj}. The value of t_h for a particular resource allocation heuristic can be determined by analyzing historic data of its execution

time. The value of t_{lj} depends on the size of input data and the data transfer rate from a shared storage to machine j. To ensure that machine idling does not occur, we let s be the maximum input data size within the set of unmapped tasks and let Λ_j be the transfer rate from a shared storage to the memory of machine j. Then, the worst case time to stage input data for any unmapped task to machine j is calculated as follows:

$$t_{lj} = \frac{s}{\Lambda_j} \tag{3}$$

We know that a free slot appears on machine queue j if a currently executing task is completed (or dropped) or a pending task is dropped because of missing its deadline. We define *earliest free slot* at time step k, denoted $e_j^{(k)}$, as the expected time that an empty slot appears on machine queue j. Let $\mathbb{E}[C(r_{1j})]$ the expected completion time of the currently executing task, and δ_{ij} the deadline of i^{th} task in machine queue j, then, $e_j^{(k)}$ is formally defined as follows:

$$e_j^{(k)} = \min\{\mathbb{E}[C(r_{1j})], \delta_{1j}, \delta_{2j}, ..., \delta_{n_j j}\} \tag{4}$$

In the scenario that the currently executing task on machine j cannot be dropped and has to complete its execution, the deadline of the currently executing task (i.e., δ_{1j}) is excluded from the definition of $e_j^{(k)}$ in Equation 4. Based on the minimum value of the earliest free slot across all machines (i.e., $\min_j(e_j^{(k)})$), we can determine the time for the next mapping event, denoted t_{map}, using Equation 5.

$$t_{map} = \min_j(e_j^{(k)}) - (t_h + t_{lj}) \tag{5}$$

If the time for the next mapping event is less than the current time, a new mapping event is scheduled immediately after the current mapping event ends.

6. Heuristics

6.1. Immediate Mode Heuristics

6.1.1. Minimum Expected Completion Time (MECT)

The Minimum Expected Completion Time (MECT) heuristic (based on the Minimum Completion Time heuristic, presented in, e.g., [4, 9, 10]) ignores the robustness of each allocation. Instead, it allocates tasks such that its expected completion time is minimized.

Using the expected execution time for each task, the expected completion time for any task r_i on machine j can be found by summing the expected execution

time of each task in $\mu_{ij}^{(k)}$ plus the time the currently executing task started. If the input queue of machine j is empty at the time of evaluation, then the expected execution time of task r_i is summed with the current time to produce a completion time. Using the expected value of the completion time for task r_i on each machine $1 \leq j \leq M$, MECT assigns task r_i to the machine that provides the earliest expected completion time.

6.1.2. Minimum Expected Execution Time (MEET)

The Minimum Expected Execution time (MEET) heuristic (based on the Minimum Execution Time heuristic, presented in, e.g., [4, 9]) also ignores the robustness of each allocation. Instead, MEET allocates each task to its minimum expected execution time machine.

In this heuristic, for each task r_i that has the expected execution time of $\mathbb{E}[r_{ij}]$ on a given machine j, we select the machine k that provides the minimum expected execution time. The MEET heuristic can be formally expressed as follows:

$$k = \min_{1 \leq j \leq M} (\mathbb{E}[r_{ij}]) \tag{6}$$

6.1.3. k-Percent Best (KPB)

The k-percent best (KPB) heuristic [4] limits the number of machines that are considered at each task assignment to the k-percent of the machines with the shortest expected execution times. Using the expected value of the execution time of task r_i on each machine, identify the k-percent of the machines that provide the shortest expected task execution times. Next, calculate the expected completion time for task r_i on each machine in the set of k-percent machines found previously and assign i to the machine that provides the earliest expected completion time.

The performance of KPB depends on the value of k. When the value of k approaches 0%, KPB performance tends to MEET. In contrast, when the value of k approaches 100%, KPB performs similar to MECT. For our simulations with eight machines, we noticed that for the scenario where a currently executing task cannot be dropped, the value of $k = 50\%$ (four of the eight machines) provides the best performance. However, for the scenario where a currently executing task is dropped as soon as misses its deadline, the value of $k = 62\%$ (five of the eight machines) provides the best performance. The reason for the lower value of k in the former scenario is that tasks are assigned to machines with shorter expected completion times. Thus, if an executing task misses its deadline, it will complete its execution quickly and a pending task can start execution.

6.1.4. MaxRobust (MR)

The MaxRobust heuristic tries to maximize the stochastic robustness of the resource allocation. In this heuristic, upon the arrival of a new task r_i at time-step $t^{(k)}$, similar to KPB, k-percent of machines with the shortest expected execution times are found. From these machines, the machine that maximizes $\rho^{(k)}$ at time-step $t^{(k)}$ is selected. That is, MaxRobust calculates the value of $\rho^{(k)}$ as if task r_i was assigned to the end of the input queue of each machine m found in the previous phase and then finds the machine that maximizes $\rho^{(k)}$. In the next step, all machines that result in $\rho^{(k)}$ within ϵ distance from the maximum $\rho^{(k)}$ are selected. From this set of machines, the one with the minimum variance [1] in completion time pmf is selected to assign task r_i.

In the simulations, for the scenario where a currently executing task cannot be dropped, we consider the value of k equal to 50% (i.e., four of the eight machines) and for the scenario where a currently executing task is dropped as soon as it misses its deadline, we consider the value of k equal to 62% (i.e., five of the eight machines). Also, the value of ϵ equal to 0.05 leads to the best performance for the MR heuristic. A higher value for ϵ leads to mapping tasks to the machines that offer lower probabilities of meeting deadlines, thus, reduces the number of tasks that can meet their deadlines.

6.2. Batch Mode Heuristics

6.2.1. Overview

In this part, we describe four batch mode resource allocation heuristics. The first three heuristics, which serve as baseline heuristics, operate in two basic phases.

In phase 1, the heuristic identifies the machine that maximizes the performance objective for each task. In phase 2, the heuristic identifies the task-machine pairing that maximizes the performance objective over all task machine pairs identified in phase 1. Details of each heuristic is described in the following subsections.

6.2.2. MinCompletion-MinCompletion (MM)

This heuristic is based on the concept from Algorithm E in [7]. In the MM heuristic, the performance objective of each phase is to minimize the expected completion time. At each mapping event $t^{(k)}$, MM first copies the batch to a separate queue Q and finds the machine j that provides the minimum expected completion time for each task in Q (phase 1). From this set of task-machine pairs, MM selects the pair that provides the overall minimum expected completion time and provisionally assigns the task to its selected machine (phase 2).

The process is repeated until all of the tasks in Q have been provisionally assigned. To complete the mapping event, MM iterates through all machine queues and for each queue where the current size is less than L (i.e., queue-size limit), tasks are moved from the provisional assignment to the available machine queue until the number of tasks in the machine queue is equal to the limit. The stopping criteria for the procedure occurs when there are no tasks left in Q, there are no remaining machines with free slots, or there are no tasks that get assigned to a machine with a free slot.

6.2.3. MinCompletion-MaxUrgency (MMU)

The MinCompletion-MaxUrgency (MMU) heuristic is also a two phase greedy heuristic that operates using expected execution times and limits the number of tasks pending completion on each machine to L. We define task urgency as the inverse of the difference between the expected completion time for the task and its deadline. Effectively, MMU emphasizes allocating tasks with the minimum slack. That is, given $\mathbb{E}[C(r_{ij})]$ the expected completion time for the task r_i on machine j, the urgency of task r_i is defined as

$$\frac{1}{\delta_i - \mathbb{E}[C(r_{ij})]} \tag{7}$$

In the first phase of MMU, the heuristic identifies the minimum expected completion time machine for each task in the batch $B(k)$. In the second phase, based on the task completion times found in phase 1, MMU selects the assignment whose task urgency is the greatest, i.e., has the smallest slack. The process is repeated as with MM.

6.2.4. MinCompletion-SoonestDeadline (MSD)

This heuristic is a variation of the two phase greedy heuristic where we favor tasks with the soonest deadline. In the first phase, the heuristic selects the machine that provides the minimum expected completion time for each task.

In the second phase, from the list of potential task-machine pairs found in the first phase, the heuristic makes the provisional assignment for the task that has the soonest deadline. In the event that two tasks have the same deadline and require the same machine, ties are broken by assigning the task that has the minimum expected completion time. The process is repeated as with MM.

6.2.5. Maximum On-time Completions (MOC)

This heuristic works based on the robustness measure proposed in this study. The procedure of MOC is shown in Figure 5. In the first step of this heuristic,

pending tasks in the machine queues whose probability of meeting their individual deadlines are less than a threshold α are dropped. In the second step, unmapped tasks that will miss their deadlines by the minimum time needed to stage their input data to any of the machines are dropped.

(1) For each machine queue j:
  (A) For each pending task r_{ij}:
    (i) If the probability of meeting deadline for task r_{ij} is less than α then
      (a) drop task r_{ij} from machine queue j.

(2) For each unmapped task u:
  (A) If sum of the current time and the minimum data staging time is greater than δ_u then
    (i) drop task u from the unmapped tasks.

(3) While stopping criteria not met do:
  (A) for each unmapped task u:
    (i) Find machine j that provides the maximum robustness for task u.
    (ii) Add task u to the list of candidate tasks for machine j.
  (B) For each machine queue j that queue-size is less than the limit:
    (i) From the list of candidate tasks for machine j, create a set of tasks that their robustnesses are within ϵ distance from the maximum robustness on machine j.
    (ii) From the set of tasks, select the task with the lowest completion time on machine j.
    (iii) Assign the selected task to machine j.
    (iv) Remove the selected task from the unmapped tasks.

Figure 5: Procedure for the MOC heuristic.

In the third step, for each unmapped task, the machine that maximizes robustness ($\rho^{(k)}$) is identified. Then, for each machine j where the number of mapped tasks is less than L, MOC identifies all the tasks within the ϵ distance from the maximum robustness for that machine. From the shortlisted tasks for each machine, the task with the minimum completion time is assigned to the machine and the remaining tasks are returned to $B(k)$. The third step continues until either there are no tasks left in $B(k)$ or there are no remaining machines with free slots.

To find the proper choice for α, we evaluated the performance of MOC with various values. We noticed that when the value of α equals to 0.2, MOC has its best performance. Lower values of α do not perform well specifically when the machine queue-size is long. Also, higher values of α leads to dropping tasks that can be completed before their deadlines. The value of ϵ equals to 0.05 results in the best performance of the MOC heuristic. Higher values of ϵ lead to mapping tasks with low probabilities of meeting deadline.

7. Simulation Setup

7.1. Overview

We perform our simulations on a limited set of machines to constrain the simulation execution times. However, our proposed methods are applicable on any larger set of machines. Our simulation environment consisted of eight machines (i.e., $M = 8$) that collectively exhibited "inconsistent" heterogeneous performance [9]; e.g., machine A may be better than machine B for task 1 but not for task 2.

In our simulation study, the task execution time distributions are assumed to be unimodal. The distributions were generated based on the Gamma distribution where the mean of the Gamma distribution was set based on execution time results for the 12 SPECint benchmark applications for a sample set of eight machines[2]. Using these distributions, we generated 500 random sample execution times for each application on each machine [40] where the scale parameter of each Gamma distribution was selected uniformly at random from the range [1,20]. After generating the sample execution times, we applied a histogram [27] to the result to produce probability mass functions that approximate the original probability density functions—one for each application on each machine. Each benchmark application served as a model for each task type to be executed by the system, creating an eight machine by twelve task type matrix of execution time pmfs. To simulate an HC system in a realistic manner, we considered a heterogeneous rate for staging the tasks' input data from a shared storage to each machine. Similar to [41], we simulated the data staging rate to each machine based on a Gamma distribution with $mean = 0.68$ Gbps and Coefficient of Variation (CV) equal to 0.1.

[2]The eight machines chosen to compose the HC suite in our simulation trials were as follows: Dell Precision 380 3Ghz Pentium Extreme Edition, Apple iMac 2Ghz Intel Core Duo, Apple XServe 2Ghz Intel Core Duo, IBM System X 3455 AMD Opteron 2347, Shuttle SN25P AMD Athlon 64 FX-60, IBM System P 570 4.7Ghz, SunFire 3800, and IBM BladeCenter HS21XM.

7.2. Generating Workload

The workload arrival rates for the trials are generated based on the systems investigated by the Extreme Scale Systems Center (ESSC) at Oak Ridge National Laboratory (ORNL) [19, 20]. Each trial includes a workload of tasks during a 10,000 time-unit period. We model an oversubscribed system that receives approximately 1,200 tasks during the period.

For performance evaluation, we need to analyze the statistical data based on an oversubscribed system. However, at the beginning and at the end of the simulation, the queues are not full and the results do not represent the condition of an oversubscribed system. Therefore, for the statistical analysis of the results, we ignore 100 tasks from the beginning and the end of the workload trials, and the remaining 1,000 tasks are utilized for statistical analysis and plotting graphs.

For generating task arrivals, we start by finding the mean arrival rate of tasks for every task type by sampling from a Gaussian distribution. The mean for this distribution is determined by dividing the desired number of tasks by the number of task types and the variance is 10% of the mean. Then, the mean arrival rate for each task type is determined by dividing the estimated number of tasks of that type into the number of time-units. The actual number and arrival times of tasks in each task type is generated by sampling from the arrival rate of that task type.

To capture the *bursty* behavior of user demand, we consider an arrival pattern that includes two types of intervals, namely, baseline and burst intervals. Comparing to the burst intervals, the baseline intervals have a lower arrival rate and last for a longer duration, while burst intervals are shorter but have higher arrival rate. The mean arrival rate for each task type is modified in each interval to generate the actual arrival rate during that interval. For each baseline interval, the arrival rate is determined by multiplying the mean arrival rate with a number uniformly sampled from the range [0.5,0.75]. To obtain the arrival rate during a burst period, the mean arrival rate is multiplied with a uniformly sampled number from the range [1.25,1.5]. The duration of each baseline interval is obtained by uniformly sampling from the range [180,300] time-units, whereas for each burst interval the range is [30,90] time-units. Figure 6 demonstrates an instance of arrival rate patterns for four task types with their baseline and burst intervals during the first 1,400 time units.

Based on the arrival rate pattern generated for every task type, the arrival time of the tasks of that type can be generated. More specifically, arrival times for tasks of a particular type are generated by sampling from an exponential distribution with the rate parameter equivalent to the arrival rate during different intervals. This means that during intervals with higher arrival rate (i.e., burst intervals) the

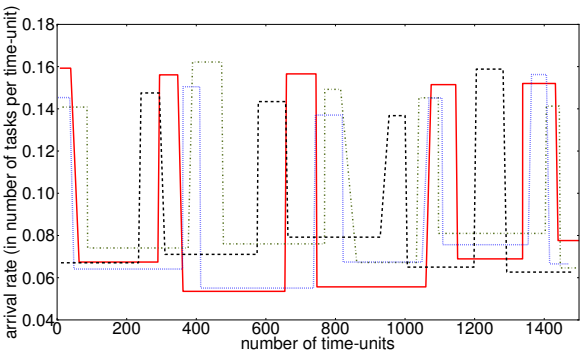


Figure 6: Arrival rate patterns with baseline and burst intervals for four task types.

sampled time from the exponential distribution is lower, hence, the arrival time of the next task is closer to the current task's arrival time.

The deadline for a given task i is generated by summing the arrival time of the task, denoted arr_i, with the average execution time of the task type on all machines, denoted avg_i, and the average execution time of all task types on all machines, denoted avg_{all}. We also consider a coefficient β for avg_{all} that enables us to loosen or tighten the generated deadlines in the workload trials. Therefore, the deadline of a given task i is calculated based on Equation 8.

$$\delta_i = arr_i + avg_i + (\beta \cdot avg_{all}) \tag{8}$$

The reason that we generate the tasks deadlines in this way is to provide a fair amount of time for each task to meet its deadline. For that purpose, in addition to the task average execution time, we provide an extra (slack) time to cover the waiting times in the queues, which can be significant in an oversubscribed system. To provide the same slack time to all tasks, the average execution time of all task types on all machines are included in generating the deadlines. In the experiments, the default value of β is 1. In addition, in Section 8.3, we investigate the impact of deadline tightness on different mapping heuristics with varying the value of β.

We expect that the size of input data for each task (e.g., size of images in an image processing system) follows a Gaussian distribution. Therefore, in workload trials, size of input data for each task is generated by sampling from a Gaussian distribution. For simulation, we have modified tasks' mean input data size to investigate the impact of this factor on the performance of different resource allocation heuristics. In each trial, the variance of the normal distribution for generating tasks' input data size is 10% of the mean.

To study the behavior of different resource allocation heuristics comprehensively, we investigate the impact of different parameters on the performance of resource allocation heuristics. More specifically, we investigate the impact of varying queue-size limits, tasks input data size, and the over-subscription level in this environment.

For all experiments we use the same task type to machine execution time pmf, the same arrival rate pattern, and the same deadline for each type (relative to the task's arrival time). Each experiment in this simulation study is carried out on 100 independent workload trials and the average and 95% Confidence Interval of the results are reported. Each simulation trial includes a new workload of tasks with different task arrivals, task types, task required input data size, and task deadlines.

8. Results and Analysis

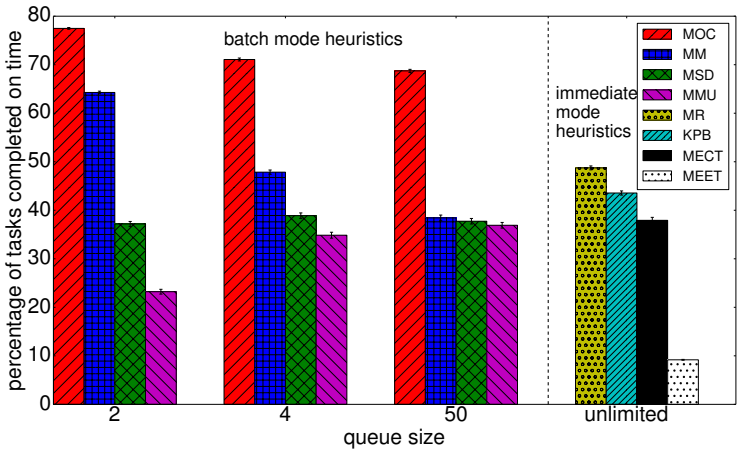
8.1. Identifying Optimal Machine Queue-Size Limit

In this experiment, we investigate the proper machine queue-size limit (L) for the batch mode resource allocation. For this purpose, we examine how different batch mode heuristics perform when the size of machine queues varies. We define *queue-size* as the pending tasks in a machine and the currently executing task in that machine. For instance, when the machine queue-size in a system is limited to 2, it means that in each machine queue there can be an executing task and a pending task waiting for execution on that machine.

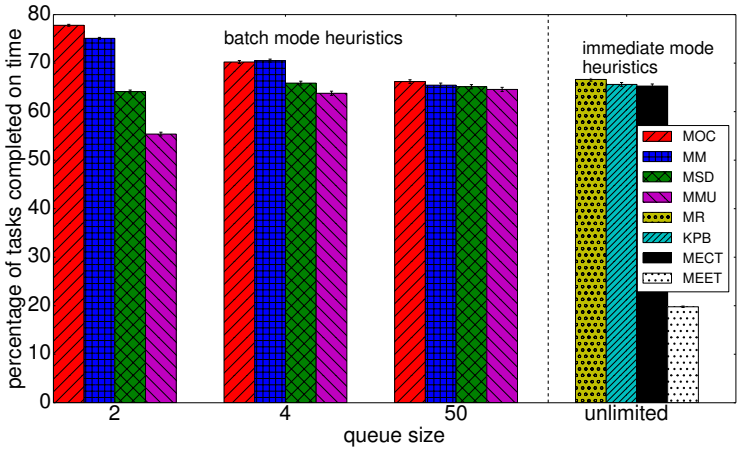
We compare the performance of different heuristics when the machines' queue-sizes are short (queue-size limits are 2 and 4) against when the queue-sizes are large (queue-size limit is 50). It is worth noting that we have evaluated other queue-sizes between these extremes to confirm our observations. However, for the sake of better presentation, we plot simulation results for the three mentioned queue-size limits. Moreover, we compare performance of the immediate mode resource allocation heuristics, that have unlimited machine queue-size, against the batch mode heuristics with various machine queue-size limits.

In this experiment, the average size of a task's input data is 64 MB and 1,000 tasks are considered for the evaluation. Our evaluation metric in this experiment is the percentage of tasks that are completed before their deadlines. Results of this experiment for the scenario where the currently executing task cannot be dropped is demonstrated in Figure 7(a) and for the scenario where the currently executing task is dropped as soon as it misses its deadline is demonstrated in Figure 7(b).

In Figure 7(a), we notice that, regardless of the queue-size limit, both the immediate and batch modes heuristics based on the defined stochastic robust mea-



(a) executing task must be completed and cannot be dropped



(b) executing task is dropped as soon as misses its deadline

Figure 7: Percentage of tasks completed on time when immediate and batch mode resource allocation heuristics with different machine queue-size limits are considered. The horizontal axis shows the different machine queue-size limits evaluated. In this experiment, the average task's input data size is 64 MB and 1,000 tasks are considered for the evaluation. Results are averaged over 100 runs and 95% confidence interval of the results are reported.

sure (i.e., MR and MOC) outperform the other heuristics. We also observe that, in general, the performance of batch mode heuristics MOC and MM is better than immediate mode heuristics. For instance, the performance of MOC when for machine queue-size 2 is on average 78% whereas the performance of MR is on average 49%. This is because batch mode resource allocation heuristics work on a group of tasks and select the best allocations. Furthermore, unallocated tasks of the batch can be re-mapped during the next mapping event along with the new arriving tasks. This provides an opportunity for the new arriving tasks to be allocated earlier than the previously arriving tasks. Whereas, in the immediate mode, the new arriving tasks always have to wait in the machine queues for tasks that have arrived earlier to exit the system.

Both Figures 7(a) and 7(b) demonstrate that shorter machine queue-sizes in MOC and MM lead to more tasks completed on time. The reason is that, when machine queue-size is short, there is less compound uncertainty in tasks completion time and, therefore, heuristics can make better allocation decisions. In contrast, we notice a slight reduction in the performance of MMU and MSD when machine queue-size is larger (e.g., when queue-size limit is 4 or 50). This is because these heuristics tend to map tasks with fast-approaching deadlines that are likely to miss their deadlines and be dropped. Therefore, when the queue-size is larger, these heuristics have the opportunity to map more tasks in a machine queue and if some of them are dropped (due to missing their deadlines) there are other ready-to-run tasks in the machine queue that can start execution.

In Figures 7(a) and 7(b), we notice that when queue-size limit is 50, there is no statistically significant difference between the percentage of tasks that are completed on time using the MM, MSD, and MMU batch mode heuristics (also called baseline heuristics). The performance of the baseline heuristics for queue-size limit 50 is on average approximately 40% and 65% in Figures 7(a) and 7(b), respectively. The reason is that when the queue-size is large, the compound uncertainty in tasks execution times increases and neutralizes the impact of resource allocation heuristics.

In Figure 7(b), we notice that the performance of immediate and most of batch mode resource allocation heuristics have remarkably increased comparing to the corresponding case in Figure 7(a). However, performance of MOC in Figure 7(b) does not vary significantly in comparison with Figure 7(a). The reason is that MOC initially drops tasks that cannot meet their deadlines and tasks that are allocated to machines are able to complete their executions before their deadlines. We can conclude that dropping of the executing tasks that have missed their deadlines, in general, favors the baseline heuristics that do not consider stochasticity in tasks

execution times in their decisions. In fact, dropping of an executing task that has missed its deadline alleviates the impact of improper allocation decisions in the baseline resource allocation heuristics.

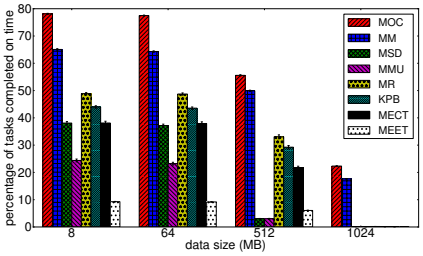
For further analysis of the experiment results, in addition to the percentage of tasks that meet their deadlines, we are interested to see how close to their deadlines the tasks are completed in different mapping heuristics. We observed that, in comparison with other heuristics, MOC and MM complete tasks far earlier than their deadlines, on average approximately 80 and 75 time units, respectively. In the case of MOC, the reason is that it maps tasks that have the highest probabilities of meeting their deadlines and therefore it is expected that they complete earlier than their deadlines. Similarly, the MM heuristic prioritizes tasks with minimum completion time, which leads to completing tasks before their deadlines. There is no statistically significant difference between other heuristics from this perspective. Also, in the scenario where executing tasks cannot be dropped, for those tasks that miss their deadlines, we analyze how much after their deadlines the tasks are completed. We observed that heuristics that function based on the robustness measure complete the tasks the earliest.

We analyzed the ratio of tasks that meet their deadlines for each task type. The results show that there is no statistically significant difference in the percentage of tasks that meet their deadlines in each task type. For instance, in the MOC heuristic, shown in Figure 7(b), when queue size is 2, between 70% and 82% of each task type meets the deadline. The same patterns occur in the other heuristics. The reason is that, the mapping heuristics we provide in this work do not discriminate tasks based on their types.

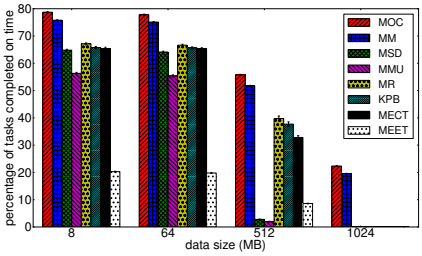
8.2. Impact of Varying Input Data Size and Connection Speed

In this experiment, we demonstrate the impact of tasks input data size on the performance of various resource allocation heuristics. To this end, we generated workload trials where the average tasks input data sizes vary from 8 MB to 1,024 MB. In this experiment, the machine queue-size is 2, and 1,000 tasks are considered for the evaluation.

Results of this experiment for the scenario where the currently executing task cannot be dropped is demonstrated in Figure 8(a) and for the scenario where the currently executing task is dropped as soon as it misses its deadline is demonstrated in Figure 8(b). According to Figures 8(a) and 8(b), as the average size of tasks' input data increases, in general, the performance of all heuristics in the immediate and batch mode decreases. For instance, performance of the MM heuristic in Figure 8(a) when the average task data size is 8 MB is on average 65% which



(a) executing task must be completed and cannot be dropped



(b) executing task is dropped as soon as it misses its deadline

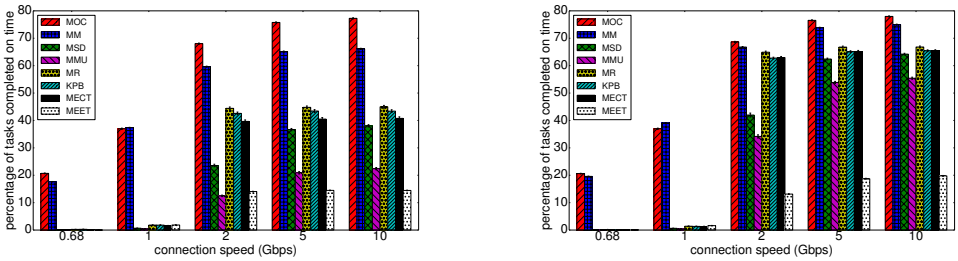
Figure 8: Percentage of tasks completed on time when immediate and batch mode resource allocation heuristics are applied and average size of tasks' required input data vary. The horizontal axis shows the different input data sizes evaluated. In this experiment, the machine queue-size limit is 2 and 1,000 tasks are considered for the evaluation. Results are averaged over 100 runs.

drops to on average 17.6% when the average data size is 1,024 MB. The reason for the performance drop is that, when the average tasks' input data size increases, a machine becomes idle while data staging for a task mapped to that machine has not been completed. To avoid such performance drop, we repeated the experiment with larger machine queue-sizes of 4 and 5 to pre-stage input data for more tasks. However, we did not observe any improvement in the performance of the heuristics. The reason is that the impact of performance degradation resulting from larger queue-sizes is more influential than the improvement resulting from pre-staging input data for more tasks.

We observe that the performance of MMU and MSD has remarkably dropped in both Figures 8(a) and 8(b), when the average tasks' input data size is 512 or 1024 MB. The reason is that these heuristics naturally tend to map tasks with fast-approaching deadlines that have a high probability of being dropped. Therefore, machines remain idle and new mapped tasks have to wait for their input data to be staged. This delay in execution leads to missing tasks deadlines, specifically when tasks' input data sizes are large.

In Figures 8(a) and 8(b), we also can notice that MOC outperforms other heuristics. Even when the average task's input data size is 1,024 MB, the MOC heuristic can complete on average 22.3% of the tasks on time, which is moderately better than MM, and significantly better than the other heuristics.

To analyze the impact of connection speed in the HC system on the performance of the mapping heuristics, in another experiment, we vary the average connection speed of the HC system from 0.68 Gbps to 10 Gbps. Results of the exper-



(a) executing task must be completed and cannot be dropped

(b) executing task is dropped as soon as it misses its deadline

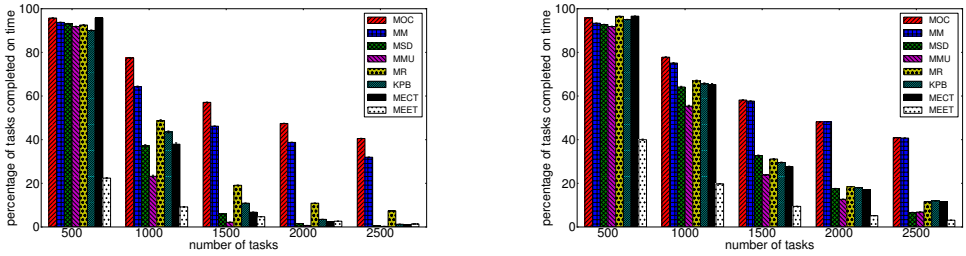
Figure 9: Percentage of tasks completed on time when immediate and batch mode resource allocation heuristics are applied and the average connection speed in the HC system varies. The horizontal axis shows the different connection speeds evaluated and the vertical axis shows the percentage of tasks that complete on time. In this experiment, the machine queue-size limit is 2, the tasks' average input data size is 1,024 MB, and 1,000 tasks are considered for the evaluation. Results are averaged over 100 runs.

iment are averaged over 100 runs and are depicted in Figure 9. In this figure, the horizontal axis shows different connection speeds and the vertical axis shows the performance of each heuristic (i.e., the percentage of tasks that meet their deadlines). In this experiment, 1,000 tasks are evaluated, the average input data for each task is 1024 MB, and the machine queue size is 2.

In both Figures 9(a) and 9(b), we observe that as the average connection speed increases, the performance of the heuristics improves. For instance, in Figure 9(a), the performance of MOC increases from 20% when the connection speed is 0.68 Gbps to approximately 75% when the connection speed is 10 Gbps. The reason is that when there is a higher connection speed it takes less time for tasks to stage their required data into the allocated machines. Therefore, the execution of the tasks is not delayed. The results indicate the impact of data dependency on the performance of HC systems, specifically when the connection speed is low. In particular, we can see that the MOC and MM heuristics perform significantly better than other heuristics, even in the presence of low connection speed. The reason is that these heuristics allocate tasks that have sufficient time to meet their deadlines. Thus, even in an HC system with a low connection speed, some tasks can complete before their deadlines.

8.3. Impact of the Over-subscription Level

In this experiment, we evaluate the impact of varying the over-subscription level on the performance of different resource allocation heuristics. To vary the



(a) executing task must be completed and cannot be dropped

(b) executing task is dropped as soon as it misses its deadline

Figure 10: Percentage of tasks completed on time using immediate and batch mode resource allocation heuristics when the over-subscription level varies in the system. The horizontal axis shows the different number of tasks evaluated. In this experiment, the average task's input data size is 64 MB and the machine queue-size limit is 2. Results are averaged over 100 runs and 95% confidence interval of the results are reported.

over-subscription level of the system, we modified the number of tasks that arrive during the same simulation period in the workload trials—from 500 to 2,500 tasks as shown in Figure 10. In this experiment, the average task input data size is 64 MB and the machine queue-size limit is 2.

Results of this experiment for the scenario where the currently executing task cannot be dropped are demonstrated in Figure 10(a) and for the scenario where the currently executing task is dropped as soon as it misses its deadline are demonstrated in Figure 10(b). According to Figure 10(a), MOC generally outperforms other batch and immediate mode heuristics. The only exception is when the HC system receives 500 tasks (i.e., when the over-subscription level is very low). In this case, the immediate mode heuristics (except MEET) perform as efficient as the batch mode heuristics. Specifically, when the system receives 500 tasks, both MOC and MECT lead to on average 95.7% performance. However, as the system becomes more oversubscribed (when there are 1,000 tasks and more) the performance of the immediate mode heuristics drops significantly in comparison to the MOC and MM heuristics. For instance, in Figure 10(a), when the system receives 2500 tasks, MOC performance is on average 40.7% whereas the performance of MECT drops to on average 1%.

The reason for the performance drop in the immediate mode heuristics is that, as the over-subscription level increases, more tasks are queued in each machine, thus, the execution of arriving tasks is delayed and ultimately they miss their deadlines. This drop in the performance shows that the immediate mode heuristics

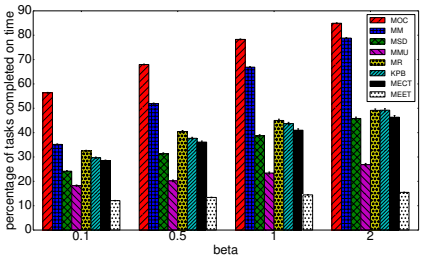
cannot cope with the increase in the over-subscription level. We can conclude that when the system is not oversubscribed, an efficient immediate mode heuristic (e.g., MR or MECT) can provide an acceptable level of performance without having the complexity of batch mode resource allocation heuristics.

In Figure 10(a), we notice that when there are more than 1,000 tasks in the system, MOC provides a better performance in comparison to the other batch or immediate mode heuristics. For instance, when there are 1,500 tasks in the system, the MOC performance is on average 60% whereas the performance of MM is 48%. This can be attributed to the fact that it tries to maximize the stochastic robustness measure. However, the reason MM performs well is that it prioritizes shorter tasks that complete more quickly.

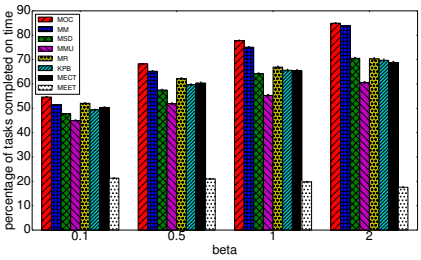
In Figure 10(a), we observe that when the over-subscription level increases to 1,500 tasks, performance of the MMU and MSD heuristics decreases sharply. The reason for this decrease is that when the over-subscription level of the HC system is high, allocating urgent tasks (i.e., tasks with tight deadlines) that cannot complete on time leads to long waiting times and missing deadlines for other tasks that are in machine queues or in the batch. However, in Figure 10(b), we observe that the performance decrease of the MMU and MSD heuristics is not as sharp as Figure 10(a). This can be attributed to the fact that the currently executing tasks that miss their deadlines are dropped and, therefore, the tasks that are waiting either in the unmapped list or in the machine queues have shorter waiting times and are able to complete on time. We can conclude that the urgency of tasks is not an appropriate measure for resource allocation, particularly in circumstances that the over- subscription level of the HC system is high.

To analyze the performance of the mapping heuristics in an oversubscribed HC system further, we conduct an experiment with various deadlines for tasks. To generate different deadlines for tasks in the workload trials, we vary the value of the β parameter in Equation 8 (see Section 7.2 for more details). As shown in the horizontal axis of Figure 11, the value of β varies from 0.1 to 2. The vertical axis shows the percentage of tasks that complete before their deadlines. In this experiment, 1,000 tasks are evaluated, the average task input data size is 64 MB, and the machine queue-size limit is 2.

As we can see in Figure 11, when the tasks deadlines increase (i.e., the value of β increases), the performance of the mapping heuristics rises, in general. We also observe in both Figures 11(a) and 11(b) that the performance difference between the two best heuristics (MOC and MM) and the other heuristics becomes more significant when the deadline is loose. The reason is that when tasks have tight deadlines, regardless of the mapping heuristic utilized, they cannot complete



(a) executing task must be completed and cannot be dropped



(b) executing task is dropped as soon as it misses its deadline

Figure 11: Percentage of tasks completed on time when immediate and batch mode resource allocation heuristics are applied and the deadline of tasks varies. Higher values of β, in the horizontal axis, shows looser deadlines for tasks. In this experiment, the machine queue-size limit is 2, the tasks' average input data size is 64 MB, and 1,000 tasks are considered for the evaluation. Results are averaged over 100 runs.

before their deadlines. However, when the deadlines are loose, the tasks have more time to complete before their deadlines, and the impact on the performance by making efficient allocation decisions in mapping heuristics such as MOC is more visible on the performance.

9. Conclusion and Future Work

In this paper, we considered the problem of dynamic resource allocation in an oversubscribed heterogeneous computing system with the goal of completing tasks before their individual deadlines and being robust against stochasticity in task execution times. The tasks are subject to individual hard deadlines and they are dropped if they cannot meet their deadlines. For a currently executing task that misses its deadline, we studied two scenarios: (a) it cannot be dropped (i.e., has to complete its execution) and (b) the currently executing task is dropped as soon as it misses its deadline. We assumed that a distribution function is available for the execution time of each task on each machine from historical data or experimentation. We investigated immediate and batch mode resource allocation schemes to map arriving tasks to machines. We proposed methods to determine the stochastic completion times in the presence of task dropping. Then, the stochastic completion time was used to define a robustness measure to quantify the performance of immediate and batch mode resource allocations. The robustness measure was utilized by the resource allocation heuristics that were proposed for the system.

We observed that MOC, which is a batch mode heuristic that operates based on the proposed robustness measure, statistically outperforms other heuristics, specifically in circumstances where the currently executing task cannot be dropped and has to be completed. This heuristic can tolerate increases in the tasks' input data size and increase in the over-subscription level better than other evaluated heuristics. In addition, we noticed that the immediate mode heuristics have acceptable performance when the over-subscription level is low. Therefore, in such circumstances, batch mode resource allocation is not recommended. However, the immediate mode heuristics could not tolerate an increase in the over-subscription level compared to the MOC and MM heuristics. We also concluded that the performance of the MOC and MM heuristics have an inverse relation with the machine queue-size in batch mode. Therefore, we chose to limit the machine queue-size to the minimum (one pending task and one task executing) in each machine.

In the future, we plan to work on resource allocation heuristics that consider the possibility of delaying the allocation of tasks to find better allocations later. Also, we are interested in implementing the stochastic robust resource allocation schemes in a real system. However, we do not have such infrastructure available at the time of this study.

Acknowledgements

The authors thank Mark Oxley and Ryan Friese for their useful comments on this work. This research was supported by the NSF under grant numbers CNS-0905399, CNS-0615170, ECCS-0700559, and CCF-1302693, and by the Colorado State University George T. Abell Endowment. This research used the CSU ISTeC Cray System supported by NSF Grant CNS-0923386.

Preliminary versions of portions of this material were presented at the IEEE International Parallel and Distributed Processing Symposium [42], International Conference on Parallel Processing [43], and International Conference on Parallel and Distributed Processing Techniques and Applications [44].

References

[1] L.-C. Canon, E. Jeannot, Evaluation and optimization of the robustness of DAG schedules in heterogeneous environments, IEEE Transactions on Parallel and Distributed Systems 21 (4) (2010) 532–546.

[2] S. Ali, A. A. Maciejewski, H. J. Siegel, J. Kim, Measuring the robustness of a resource allocation, IEEE Transactions on Parallel and Distributed Systems 15 (7) (2004) 630–641.

[3] J. Smith, V. Shestak, H. J. Siegel, S. Price, L. Teklits, P. Sugavanam, Robust resource allocation in a cluster based imaging system, Parallel Computing 35 (7) (2009) 389–400.

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

[5] R. F. d. Silva, W. Chen, G. Juve, K. Vahi, E. Deelman, Community resources for enabling research in distributed scientific workflows, in: Proceedings of the 10th IEEE International Conference on e-Science, 2014, pp. 177–184.

[6] E. Coffman, J. Bruno, Computer and Job-shop Scheduling Theory, New York, NY: John Wiley & Sons, 1976.

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

[8] K. Kaya, B. Uçar, C. Aykanat, Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories, Journal of Parallel and Distributed Computing 67 (3) (2007) 271–285.

[9] T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, R. F. Freund, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, Journal of Parallel and Distributed Computing 61 (6) (2001) 810–837.

[10] V. Yarmolenko, J. Duato, D. K. Panda, P. Sadayappan, Characterization and enhancement of dynamic mapping heuristics for heterogeneous systems, in: Proceedings of the International Workshop on Parallel Processing, ICPP '00, 2000, pp. 437–444.

[11] I. Al-Azzoni, D. G. Down, Dynamic scheduling for heterogeneous desktop grids, Journal of Parallel and Distributed Computing 70 (12) (2010) 1231–1240.

[12] Y. Chen, H. C. Liao, T. Tsai, Online real-time task scheduling in heterogeneous multicore system-on-a-chip, IEEE Transactions on Parallel and Distributed Systems 24 (1) (2013) 118–130.

[13] V. Shestak, J. Smith, A. A. Maciejewski, H. J. Siegel, Stochastic robustness metric and its use for static resource allocations, Journal of Parallel and Distributed Computing 68 (8) (2008) 1157–1173.

[14] Y. Lee, A. Y. Zomaya, A novel state transition method for metaheuristic-based scheduling in heterogeneous computing systems, IEEE Transactions on Parallel and Distributed Systems 19 (9) (2008) 1215–1223.

[15] Y. C. Lee, A. Y. Zomaya, Rescheduling for reliable job completion with the support of clouds, Future Generation Computer Systems 26 (8) (2010) 1192–1199.

[16] Digitalglobe Inc., `http://www.digitalglobe.com` (accessed July 20, 2013).

[17] G. Teodoro, T. Pan, T. M. Kurc, J. Kong, L. A. D. Cooper, N. Podhorszki, S. Klasky, J. H. Saltz, High-throughput analysis of large microscopy image datasets on CPU-GPU cluster platforms, in: Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium, IPDPS '13, 2013, pp. 103–114.

[18] L. V. Fulton, L. S. Lasdon, R. R. McDaniel, M. N. Coppola, Two-stage stochastic optimization for the allocation of medical assets in steady-state combat operations, The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology 7 (2) (2010) 89–102.

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

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

[21] B. Khemka, R. Friese, S. Pasricha, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, S. Powers, M. Hilton, R. Rambharos, S. Poole, Utility driven dynamic resource management in an oversubscribed energy-constrained heterogeneous system, in: Proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium Workshops, IPDPSW '14, 2014, pp. 58–67.

[22] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph, A. Ganjam, J. Zhan, H. Zhang, Understanding the impact of video quality on user engagement, in: Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM '11, 2011, pp. 362–373.

[23] S. Hengstler, D. Prashanth, S. Fong, H. Aghajan, Mesheye: A hybrid-resolution smart camera mote for applications in distributed intelligent surveillance, in: Proceedings of the 6th International Conference on Information Processing in Sensor Networks, IPSN '07, 2007, pp. 360–369.

[24] G. Sharp, N. Kandasamy, H. Singh, M. Folkert, GPU-based streaming architectures for fast cone-beam CT image reconstruction and demons deformable registration, Physics in Medicine and Biology 52 (19) (2007) 5771–5783.

[25] N. Guan, W. Yi, Q. Deng, Z. Gu, G. Yu, Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling, Journal of System Architecture 57 (5) (2011) 536–546.

[26] X. Qin, H. Jiang, A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems, Parallel Computing 32 (5) (2006) 331–356.

[27] L. Wasserman, All of Statistics: A Concise Course in Statistical Inference, New York, NY: Springer Science+Business Media, 2005.

[28] Y. A. Li, J. K. Antonio, H. J. Siegel, M. Tan, D. W. Watson, Determining the execution time distribution for a data parallel program in a heterogeneous computing environment, Journal of Parallel and Distributed Computing 44 (1) (1997) 35–52.

[29] L. Bölöni, D. C. Marinescu, Robust scheduling of metaprograms, Journal of Scheduling 5 (5) (2002) 395–412.

[30] CSU Information Science and Technology Center. ISTeC Cray High Performance Computing (HPC) System, http://istec.colostate.edu/activities/cray (accessed Dec. 21, 2014).

[31] A. Dogan, F. Ozguner, Genetic algorithm based scheduling of meta-tasks with stochastic execution times in heterogeneous computing systems, Cluster Computing 7 (2) (2004) 177–190.

[32] A. Kumar, R. Shorey, Performance analysis and scheduling of stochastic fork-join jobs in a multicomputer system, IEEE Transactions on Parallel and Distributed Systems 4 (10) (1993) 1147–1164.

[33] J. Cao, K. Li, I. Stojmenovic, Optimal power allocation and load distribution for multiple heterogeneous multicore server processors across clouds and data centers, IEEE Transactions on Computers 63 (1) (2014) 45–58.

[34] C. Delimitrou, C. Kozyrakis, QoS-aware scheduling in heterogeneous datacenters with Paragon, ACM Transactions on Computer Systems 31 (4) (2013) 1–34.

[35] I. Al-Azzoni, D. G. Down, Linear programming-based affinity scheduling of independent tasks on heterogeneous computing systems, IEEE Transactions on Parallel and Distributed Systems 19 (12) (2008) 1671–1682.

[36] K. Li, X. Tang, K. Li, Energy-efficient stochastic task scheduling on heterogeneous computing systems, IEEE Transactions on Parallel and Distributed Systems 25 (11) (2014) 2867–2876.

[37] Y. Xu, K. Li, T. T. Khac, M. Qiu, A multiple priority queueing genetic algorithm for task scheduling on heterogeneous computing systems, in: Proceedings of the 14th IEEE International Conference on High Performance Computing and Communication, HPCC '12, 2012, pp. 639–646.

[38] B. D. Young, J. Apodaca, L. D. Briceño, J. Smith, S. Pasricha, A. A. Maciejewski, H. J. Siegel, B. Khemka, S. Bahirat, A. Ramirez, Y. Zou, Deadline and energy constrained dynamic resource allocation in a heterogeneous computing environment, The Journal of Supercomputing 63 (2) (2013) 326–347.

[39] A. Leon-Garcia, Probability & Random Processes for Electrical Engineering, Reading, MA: Addison Wesley, 1989.

[40] S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, S. Ali, Representing task and machine heterogeneities for heterogeneous computing systems, Tamkang Journal of Science and Engineering Special Tamkang University 50th Anniversary Issue, 3 (3) (2000) 195–208, invited.

[41] B. Sotomayor, R. S. Montero, I. M. Llorente, I. Foster, Resource leasing and the art of suspending virtual machines, in: Proceedings of the 11th IEEE International Conference on High Performance Computing and Communications, HPCC '09, 2009, pp. 59–68.

[42] J. Smith, L. Briceño, A. A. Maciejewski, H. J. Siegel, T. Renner, V. Shestak, J. Ladd, A. Sutton, D. Janovy, S. Govindasamy, A. Alqudah, R. Dewri, P. Prakash, Measuring the robustness of resource allocations in a stochastic dynamic environment, in: Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium, IPDPS '07, 2007, pp. 1–10.

[43] J. Smith, E. K. P. Chong, A. A. Maciejewski, H. J. Siegel, Stochastic-based robust dynamic resource allocation in a heterogeneous computing system, in: Proceedings of the 38th International Conference on Parallel Processing, ICPP '09, 2009, pp. 188–195.

[44] J. Smith, J. Apodaca, A. A. Maciejewski, H. J. Siegel, Batch mode stochastic-based robust dynamic resource allocation in a heterogeneous computing system, in: Proceedings of the 16th International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '10, 2010, pp. 263–269.