J Supercomput (2013) 63:326–347
DOI 10.1007/s11227-012-0740-7

Deadline and energy constrained dynamic resource allocation in a heterogeneous computing environment

B. Dalton Young · Jonathan Apodaca · Luis Diego Briceño · Jay Smith · Sudeep Pasricha · Anthony A. Maciejewski · Howard Jay Siegel · Bhavesh Khemka · Shirish Bahirat · Adrian Ramirez · Yong Zou

Published online: 2 February 2012

© Springer Science+Business Media, LLC 2012

B.D. Young (✉) · L.D. Briceño · J. Smith · S. Pasricha · A.A. Maciejewski · H.J. Siegel ·
B. Khemka · S. Bahirat · A. Ramirez · Y. Zou
Department of Electrical & Computer Engineering, Colorado State University, Fort Collins, CO, USA
e-mail: dalton.young@colostate.edu

L.D. Briceño
e-mail: ldbricen@colostate.edu

J. Smith
e-mail: jtsmith@digitalglobe.com

S. Pasricha
e-mail: sudeep@colostate.edu

A.A. Maciejewski
e-mail: aam@colostate.edu

H.J. Siegel
e-mail: hj@colostate.edu

B. Khemka
e-mail: bhavesh.khemka@colostate.edu

S. Bahirat
e-mail: shirish.bahirat@colostate.edu

A. Ramirez
e-mail: adrian.ramirez@colostate.edu

Y. Zou
e-mail: yong.zou@colostate.edu

J. Apodaca · S. Pasricha · H.J. Siegel
Department of Computer Science, Colorado State University, Fort Collins, CO, USA

J. Apodaca
e-mail: jonathan.apodaca@colostate.edu

J. Smith
DigitalGlobe, Longmont, CO, USA

Abstract Energy-efficient resource allocation within clusters and data centers is important because of the growing cost of energy. We study the problem of energy-constrained dynamic allocation of tasks to a heterogeneous cluster computing environment. Our goal is to complete as many tasks by their individual deadlines and within the system energy constraint as possible given that task execution times are uncertain and the system is oversubscribed at times. We use Dynamic Voltage and Frequency Scaling (*DVFS*) to balance the energy consumption and execution time of each task. We design and evaluate (via simulation) a set of heuristics and filtering mechanisms for making allocations in our system. We show that the appropriate choice of filtering mechanisms improves performance more than the choice of heuristic (among the heuristics we tested).

Keywords Dynamic resource allocation · Heterogeneous computing · Power aware computing

1 Introduction and problem statement

Energy consumption of servers and data centers is a growing concern (e.g., [11, 19]). Some studies predict that the annualized cost of powering and cooling servers may soon exceed the annualized cost of equipment acquisition [20], which could force some servers to operate under a constraint on the amount of energy used to complete workloads.

In this research, we study the problem of dynamically allocating a collection of independent tasks to a heterogeneous computing cluster (heterogeneous both in terms of performance and power efficiency) while considering energy. We assume that the system is often oversubscribed, as is the case for the Oak Ridge National Labs Extreme Scale Systems Center system under development [13]. The goal is to maximize the number of tasks completed by their individual deadlines under a constraint on the total amount of energy used by the system. Our problem formulation is more complex than earlier approaches because we consider the combination of a heterogeneous cluster, a time-varying arrival rate for tasks that causes the system to be oversubscribed at times, tasks with individual deadlines, stochastic task execution times, an energy constraint to process a fixed number of tasks, a system model that allows the selective cancellation of some tasks, and using the concept of robustness (described in Sect. 4) in the objective functions of some heuristics.

We approach this problem by deriving resource allocation heuristics and filtering mechanisms that are capable of leveraging the cluster heterogeneities to maximize the number of tasks completed under a given energy constraint. We then compare these heuristics via simulation. Two of our heuristics are adapted from the literature to our environment, while the third is a novel heuristic that attempts to balance each task's energy consumption and probability of completing by its deadline. Additionally, our two filter mechanisms can be generically applied to any heuristic to add energy-awareness and/or robustness-awareness. Our workload (described in Sect. 3.2) consists of a dynamically-arriving mix of different task types (e.g., compute-intensive, memory-intensive). Our system is oversubscribed at times, so we cannot utilize certain energy-conserving techniques (described in Sect. 3.1). Thus, our heuristics and filters are limited to controlling energy consumption via task-to-machine mapping and processor Dynamic Voltage and Frequency Scaling (*DVFS*).

In this paper, we make the following contributions: (a) we develop a model of robustness for this environment and validate its use in allocation decisions, (b) we present an adaptation of two existing heuristics to utilize robustness and account for an energy constraint while making task-to-machine assignments, (c) we present a new heuristic for use in our environment, and (d) we demonstrate the utility of our generalized filter mechanisms via simulations, which show at least a 10% improvement in each heuristic due to filtering.

The remainder of this paper is organized as follows. The next section discusses a sampling of the most closely-related work. In Sect. 3, we define the model of the compute cluster, workload, and energy consumption of the system. Based on this system model, we formally introduce the concept of robustness and derive a robustness measure suitable to this environment in Sect. 4. Section 5 describes the heuristics used in this study. Section 6 then discusses our simulation setup, while Sect. 7 presents and analyzes our simulation results. We conclude with Sect. 8, wherein we discuss extensions to this research and future directions.

2 Related work

The problem of mapping dynamically-arriving tasks under an energy constraint is addressed in [17]. That work focused on conserving battery life in an ad-hoc grid environment while completing as many high-priority tasks as possible, followed by as many low-priority tasks as possible. The environment in [17] also used a bursty environment with oversubscription. However, our study is fundamentally different because we work with probability distributions representing uncertain task execution times, whereas the work in [17] used scalar task execution times. Also, our study focuses on a cluster environment with a single energy constraint, while the work in [17] focused on an ad-hoc grid with energy constraints on a per-component basis.

The research in [30] uses Dynamic Voltage and Frequency Scaling (*DVFS*) within a real-time system where the tasks have uncertain execution times. Unlike our study, there is no energy constraint and the system is not oversubscribed. The work in [30] also emphasizes the benefits of inter-task DVFS to take advantage of slack time, but our system is oversubscribed at times. Similarly, the research in [32] attempts to maximize a mathematical model of Quality of Service (*QoS*) under an energy constraint, but it does so using DVFS to take up slack time in an undersubscribed system, whereas our system is oversubscribed some of the time.

In [9], the authors use a multipart solution to save energy in a dynamic, real-time system with periodic tasks. Like [9], our environment is dynamic, but it is also oversubscribed. Additionally, we have the goal of completing as many tasks by their deadlines as possible under an energy constraint, whereas the study in [9] has the goal of minimizing energy under the constraint of completing all tasks by their hard deadlines. Also, the solution in [9] utilizes a static component to develop its schedules, while our heuristics are limited to immediate-mode operation (tasks are mapped immediately upon arrival [24]).

Similarly, in[18] a set of independent tasks with individual deadlines are dynamically allocated to a cluster while attempting to conserve energy. While [18] attempts to optimize the energy consumed under the constraint of completing all tasks by their deadlines, our environment has an energy constraint and we optimize the number of tasks completed. The work in[18] uses deterministic task execution times and constant arrival rates with an undersubscribed system, while our research focuses on stochastic task execution times and a bursty arrival rate with the system oversubscribed during task-arrival bursts.

This group has previously studied dynamic resource allocation in [26, 27]. This research uses some heuristics from [27] and part of the robustness definition from [26]. However, neither of these previous works deal with energy-aware scheduling. We have also studied the static allocation of independent tasks with a common deadline to optimize the energy consumed in [8].

3 System model

3.1 Cluster configuration

Our model of a cluster allows the performance and power efficiency of each node in the cluster to vary substantially. That is, the system is heterogeneous because it may consist of compute nodes that are quite different from one another. *Machine performance* is defined in terms of the time required to execute a given task, i.e., a higher performance machine will execute a task in less time than a lower performance machine. The machine performance of the nodes in this cluster is assumed to be *inconsistent* [7], i.e., because machine A is faster than machine B on one task does not imply that machine A is faster for all tasks.

Our model assumes that a cluster consists of N heterogeneous compute nodes, each with a different number of multicore processors, different number of cores in each multicore processor, different set of available processor frequencies, different power consumption profile, and different power supply efficiency. Each compute node i consists of $n(i)$ multicore processors. Each multicore processor in compute node i has $c(i)$ cores, where $c(i)$ is constant within each node. Figure 1 shows the hierarchy of nodes, multicore processors, and cores within our cluster.

We assume that each core k in multicore processor j processes tasks independently of the other cores, and all cores and multicore processors within a given compute node are homogeneous. We also assume that our cores are non-multitasking and execute one task at a time, as is the case with the ISTeC Cray XT6m system currently in use at Colorado State University [14]. In this study, we limit the size of our cluster (see Sect. 6 for details) to limit our simulation execution times, but our proposed techniques can be easily extended to larger clusters of nodes.

Our hierarchical cluster model is directly applicable to the ISTeC Cray XT6m. While the current system is homogeneous across compute nodes, there are plans to add GPUs to a limited subset of the compute nodes as well as to continually grow the system with new compute nodes based on the technology available at the time of purchase. Thus, this system will indeed be heterogeneous and follow our hierarchical model.

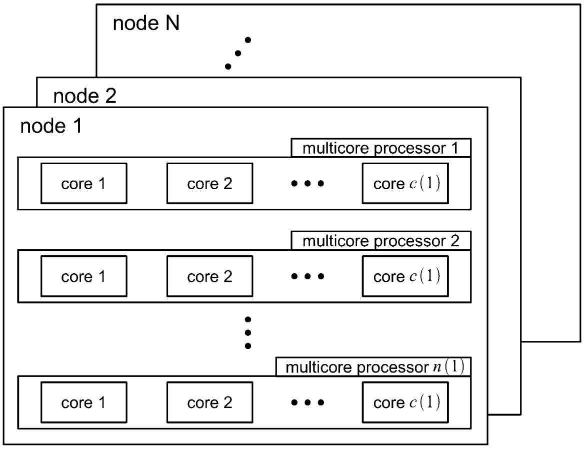


Fig. 1 The hierarchy of nodes, multicore processors, and cores that are used in our system model

Our model of available processor frequencies and processor power consumption is based on the ACPI standard [1]. The standard defines *P-states*, which are processor performance states that allow the processor to save power while executing instructions at the expense of decreased performance, i.e., increased task execution times. These states are used in DVFS implementations in many commodity processors (e.g., [3, 15]).

Given the large number of cores in the latest compute nodes (24 cores per compute node in our ISTeC Cray XT6m) and the fact that our system can be oversubscribed at any time in an unpredictable manner, turning the power off to a compute node has a high associated overhead for the systems we are considering, and is therefore not considered in our model (but it may be considered in future work). Thus, we assume that the variation in energy consumed by the shared system components of each node (such as disk drives and fans) is small compared to the energy consumption of cores and can therefore be approximated as constant and excluded from our computations (subtracted from the energy constraint before any tasks are scheduled). In this way, we are assuming that only the P-states can be used to save power (we discuss future work modifications to our model to consider power usage by memory and shared system components in Sect. 8). Although there are many different models of energy consumption in the literature, we feel that this model captures most of the salient attributes of compute node energy usage that we can control, and future work could consider extensions to the model such as including memory energy consumption and ACPI G-states. Our model can also be extended to deal with other tasks that do not benefit from DVFS (e.g., memory-intensive, communication-intensive), as described as future work in Sect. 8.

The ACPI standard defines up to 16 P-states, but we will assume that a set of only five, denoted P, is available: P_0, P_1, P_2, P_3, and P_4. Each P-state is associated with a certain clock speed and voltage configuration for a core that defines its corresponding power consumption. We will assume that, as power supplied to a core is increased, the performance of the core also will increase. We also will assume that task performance is linearly proportional to processor frequency, and proportional to processor power consumption as described in Sect. 6. Following convention [1, 4], let P_0 correspond to the base P-state that provides the highest power consumption (and therefore highest performance), and P_4 correspond to the P-state that provides the lowest power consumption (and therefore lowest performance). Power consumption in real systems can vary within a given P-state. For this study, we make the simplifying assumption that the power consumption of a core operating in a given P-state can be approximated by a scalar constant that represents the average power consumption. The power consumed by P-state π in any core (because we assume that all cores and multicore processors within a compute node are identical) of compute node i is denoted $\mu(i,\pi)$. We will discuss the values of $\mu(i,\pi)$, as well as the relative performance of the cores in each P-state, in Sect. 6.

In our environment, the resource management system controls the P-states of each core individually. The operating system of each compute node provides a power management kernel that controls the P-state transitions for each core within each multicore processor, and we assume that cores within a multicore processor can switch P-states independently [4]. In this research, we assume that there is a cluster resource manager integrated with the operating system power manager so that the cluster resource manager can direct the power management kernel to change P-states, and that P-state transition times can be ignored because they are small (hundreds of microseconds [4]) with respect to task execution times (e.g., thousands of milliseconds). Because our scheduler operates with stochastic task execution times and performs convolutions, we reduce its overhead and complexity by assuming that cores can only change P-states when idle, i.e., P-states cannot be transitioned during task execution. The cluster resource manager will execute our resource allocation heuristics and take responsibility for controlling the power consumption of the cluster—in addition to assigning tasks to cores for execution.

Within a node of the cluster, power efficiency relates the total power provided by a power supply to the actual power it consumes. For example, a power supply with 90% efficiency supplies 90 watts of power to the elements of the node for every 100 watts of power it consumes. We denote the power efficiency of the power supply in node i as $\epsilon(i)$.

3.2 The workload

The workload in the environment in our model is a dynamically-arriving collection of independent tasks, i.e., a "bag of tasks" [16] where the exact task mix is unknown prior to its arrival. However, each task is selected from a finite collection of well-known task types (such as may be found in many military, lab, government, and industrial environments), and the execution time probability mass function (*pmf*) of each task type can be derived from histograms of experiments or past runs of tasks of the given type over typical data sets (one example of such tasks can be found in [12]). The different task types are primarily compute-intensive.

In the workload, the execution time of each task is considered stochastic due to factors such as varying input data or cache conflicts, and that we are provided an execution-time pmf for each task type executing on a single core of each node in each P-state (such pmfs may in practice be obtained by historical, experimental, or analytical techniques [23, 29]). More specifically, we model the execution time of each task type as a random variable and assume that we are given a probability mass function describing the possible execution time values and their probabilities for each task type, core, and P-state combination. We also assume that power consumption is a function of P-state and processor. We will discuss the creation of execution time distributions for our simulations in Sect. 6.

Our workload is characterized by a bursty arrival rate [22], which will cause the system to be oversubscribed during burst periods and undersubscribed at other times. We assume that we are provided a deadline for completing each task z, denoted $\delta(z)$, i.e., $\delta(z)$ defines a constraint on the completion time of task z. In a real system, task deadlines can come from multiple sources (e.g., limits set by system administrators, user requirements for timely data). Each deadline is considered a hard deadline, and there is no value in completing a task after its deadline has passed (i.e., the task is not counted as completed if its deadline is missed). This is similar to the system under development at the Extreme Scale Systems Center at Oak Ridge National Lab [13]. We assume our cluster resource manager cannot stop a task after it has been scheduled and must execute it to completion as a best-effort basis even if the task misses its deadline because allowing the scheduler to stop tasks complicates our mathematical model considerably and results in a more complex scheduler (this is potential future work). We will describe our deadline and arrival rate models in more detail in Sect. 6.

We test our heuristics over a workload consisting of a window of 12 hours worth of tasks generated as described in Sect. 6. We assume that our resource management heuristics can make allocation decisions that take into account the amount of time remaining in the 12 hour window. It is assumed that processors report to the resource manager whenever a task completes, so that the resource manager will know what tasks are still awaiting execution for each core.

We limit our cluster resource manager to operating in an immediate-mode [24]. Additionally, we assume that tasks cannot be reassigned, either to a new core or a new P-state, once they are mapped, but that the resource manager may leave a task unassigned or opt to cancel a task completely just before the task begins execution. Task mapping is controlled by the resource allocation heuristic employed by the cluster resource manager.

3.3 Energy consumption

As the tasks are independent and cores can change their P-states independently of one another, we can find the energy required by each core throughout the entire simulation independently of the other cores (recall that disk and memory energy requirements are treated as a constant and are therefore not included here). We can then find the total energy required by the cluster by summing the energy required by all cores. Because we assume that cores cannot be turned off, the energy consumption for each core can be found by identifying the time of each P-state transition, calculating the time difference between successive transitions, and multiplying this time difference by the power required to support that P-state. Each core implicitly transitions to P-state P_4 whenever it is idle.

Let $\nu(i, j, k)$ denote the list of P-state transitions that are scheduled for core k of multicore processor j in node i throughout the execution of the workload, let

$|\nu(i,j,k)|$ denote the size of list $\nu(i,j,k)$, let $\nu(i,j,k,n)$ denote the nth P-state transition in list $\nu(i,j,k)$, let $\mathit{time}(x)$ denote the time of P-state transition x, and let $\mathit{pstate}(x)$ denote the ending P-state for the transition. We assume that each core makes at least two P-state transitions: one at the start of workload execution and one at the end of workload execution. If we define $\Delta t(n) = \mathit{time}(\nu(i,j,k,n)) - \mathit{time}(\nu(i,j,k,n-1))$, we can compute the energy used by each core, which we denote $\eta(i,j,k)$, as

$$\eta(i,j,k) = \sum_{n=1}^{|\nu(i,j,k)|} \mu\big(i, \mathit{pstate}\big(\nu(i,j,k,n)\big)\big) \times \Delta t(n). \qquad (1)$$

Using the energy needed by each core ($\eta(i,j,k)$) from (1), we can find the energy required to complete the entire workload, which we denote ζ, as

$$\zeta = \sum_{i=1}^{N} \sum_{j=1}^{n(i)} \sum_{k=1}^{c(i)} \frac{\eta(i,j,k)}{\epsilon(i)}. \qquad (2)$$

We let $\zeta_{\max}$ denote the energy constraint for completing the workload. Any tasks not completed within the energy constraint are considered as having missed their deadlines.

4 Robustness

4.1 Overview

We use random variables to model task execution times because the actual execution time of each task is uncertain [28]. We want our resource allocations to be "robust," meaning that they mitigate the impact of uncertainties on our performance objective [6]. More specifically, we want our resource allocations to mitigate the impact of the uncertain task execution times on our objective of completing as many tasks as possible, by their individual deadlines, within our energy constraint. This research builds on the robustness model presented in [26].

When a system is described as robust, three questions must be answered [5]: (1) What behavior makes the system robust? (2) What uncertainties is the system robust against? (3) How is the robustness of the system quantified? In our system model, an allocation is robust if it can complete all tasks by their individual deadlines; an allocation is robust against uncertainties in task execution times; and the robustness of an allocation is quantified as the expected number of tasks which will complete by their deadlines.

4.2 Stochastic task completion time

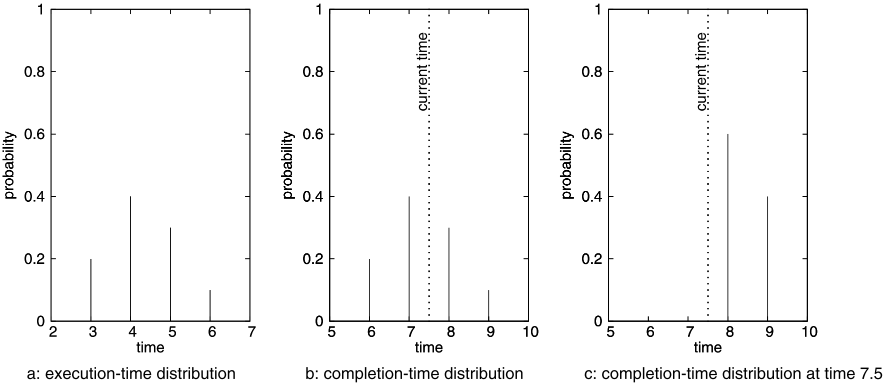
At the lth time-step t_l, we want to predict the completion time of a task z if it is assigned to a core k in multicore processor j of node i. Calculating this completion time requires combining the execution times for all tasks that have been assigned to core k with the execution time of z. When using a deterministic (i.e., nonprobabilistic) model, we calculate the completion time as the sum of the estimated execution times for all tasks assigned to core k, the estimated execution time for task z if assigned to core k, and the ready time of core k. Because we are using a stochastic model (task execution times are represented by pmfs), we calculate the completion time as the sum of the random variables represented by the pmfs and the ready time. This completion time sum requires a convolution of pmfs [21, 25]. Convolutions can take considerable time related to the total number of impulses in each convolution, which increases with each subsequent convolution. The overhead can be negligible if task execution times are sufficiently long. Furthermore, the performance gained justifies their usage.

Let $Q(t_l)$ be the set of all tasks that are either queued for execution or are currently executing on any of the cores in the cluster at time-step t_l. To determine the completion time of task z if assigned to core k of multicore processor j on node i at time-step t_l, we first identify the ordered list of tasks in $Q(t_l)$ that are assigned to core k, in order of their assignment to core k, and we let $Q(i, j, k, t_l)$ denote this list. If there are no tasks assigned to core k, then $Q(i, j, k, t_l) = \emptyset$, and the ready time of this core is equal to the current time. In this case, the stochastic completion time of task z if assigned to core k is represented by the execution-time pmf of task z on core k in its chosen P-state, shifted by the current time.

If $Q(i, j, k, t_l) \neq \emptyset$, then the execution time pmf for the currently executing task a on core k (the first task in $Q(i, j, k, t_l)$) requires additional processing prior to its convolution with the pmfs of the queued tasks (other tasks in $Q(i, j, k, t_l)$). If a began execution at time-step t_h ($h < l$), some of the impulse values of the pmf describing the completion time of a are in the past. Therefore, accurately describing the completion time of task a at time t_l requires shifting the execution-time distribution for task a by t_h (effectively creating the completion-time distribution for task a if it started execution at time t_h), removing the past impulses from the pmf (those impulses which occur at time less than t_l), and renormalizing the remaining distribution [27]. After renormalization, the resulting distribution describes the completion time of a on core k as predicted at time-step t_l. This distribution is then convolved with the execution time pmfs of the tasks in $Q(i, j, k, t_l)$ and the execution-time pmf of task z on this core in its chosen P-state to produce the completion-time pmf for task z if assigned to core k at the current time-step t_l. Figure 2 shows an example of the computation required to find the completion time distribution for a currently executing task at a specific time.

The above computations can be applied to any task in $Q(i, j, k, t_l)$ to obtain a completion-time distribution. The completion-time distribution for the currently-executing task a on core k, for example, can be found by shifting the execution-time distribution of a assigned on core k by its start time, removing impulses at time values less than t_l, and renormalizing the distribution. The completion-time distribution for task b assigned immediately after task a on core k can then be found by convolving the execution-time distribution of task b assigned on core k with the completion-time distribution of task a.

In this environment, stochastic completion time must also be modified to reflect the scheduler's ability to cancel tasks when they become ready to execute. To account for this, we can exclude from the completion-time distribution computations



a: execution-time distribution

b: completion-time distribution

c: completion-time distribution at time 7.5

Fig. 2 Example of the additional processing required at time 7.5 for a currently executing task that started at time 3. In **a**, we have the *execution*-time distribution of the task. In **b**, the *execution*-time distribution has been shifted by its start time (3) to form a *completion*-time distribution. In **c**, those impulses that occur before the current time have been removed, and the remaining distribution has been renormalized to form the *completion*-time distribution at time 7.5

those distributions for tasks which we think will be canceled (based on the current state of the system). Using the previous example, imagine we want the completion-time distribution for a task c assigned after a queued task b and an executing task a. If the scheduler is canceling any task with a probability of less than or equal to 0% of completing by its deadline, and the probability of b completing by its deadline, as calculated from its completion-time distribution, is less than or equal to 0%, then we can assume that b will be canceled and compute the completion-time distribution for task c as the convolution of the completion-time distribution for task a and the execution-time distribution for task c. It is possible that a task's probability of completion increases or decreases as it moves through the queue ($Q(i, j, k)$).

4.3 Robustness calculation

The robustness of a resource allocation in this environment is defined at a given time-step t_l as the expected number of tasks that will complete by their individual deadlines, predicted at t_l [26]. We denote this value $\rho(t_l)$. Because there is no inter-task communication in our cluster environment (all tasks are independent), the expected number of on-time task completions for all tasks assigned to a common core k of multicore processor j in node i (all tasks in $Q(i, j, k, t_l)$) is independent across all cores and nodes. We let $\rho(i, j, k, t_l)$ denote this value. If we let $\rho(i, j, k, \pi, t_l, q)$ denote the probability of task z in $Q(i, j, k, t_l)$ with P-state π finishing by its deadline $\delta(z)$, then we can compute $\rho(i, j, k, t_l)$ as

$$\rho\,(i, j, k, t_l) = \sum_{\forall z \in Q(i,j,k,t_l)} \rho\,(i, j, k, \pi, t_l, z)\,. \tag{3}$$

We can therefore calculate $\rho\,(l)$ as

$$\rho\,(t_l) = \sum_{i=1}^{N} \sum_{j=1}^{n(i)} \sum_{k=1}^{c(i)} \rho\,(i, j, k, t_l)\,. \tag{4}$$

To complete as many tasks as possible by their individual deadlines, the research in [26] indicates that we should maximize the expected number of on-time completions ($\rho(t_l)$) at each time-step [26]. However, our system is limited to immediate-mode mapping. Therefore, if we are assigning a task z at time-step t_l, we can maximize $\rho(t_l)$ by assigning z to the core k of multicore processor j in node i and P-state π that maximizes $\rho(i, j, k, \pi, t_l, z)$ (assigning the task where it has the highest probability of completing by its deadline).

To find a task's probability of completing by its deadline given a node, multicore processor, core, and P-state assignment, we can merely find the completion-time distribution for the assignments as described in Sect. 4.2, and then sum the impulses in the distribution that are less than the deadline. We will use this number in our filters and heuristics.

5 Heuristics and filters

5.1 Overview

In this study, we adapted two task-scheduling heuristics taken from the literature to our cluster environment. We also created a new heuristic and implemented a random-assignment heuristic. An *assignment* consists of mapping a single task to a node, multicore processor, core, and P-state. Each heuristic, operating in an immediate-mode, assigns a single task to a node, multicore processor, core, and P-state for execution. We developed two filtering mechanisms that can be generically applied to any task-scheduling heuristic to limit the set of feasible assignments the heuristic may use. A filtering mechanism could eliminate all potential assignments, which would cause the task to remain unassigned and be discarded. We also developed a generic start-time cancellation mechanism that can allow any task-scheduling heuristic to take advantage of the system's ability to cancel tasks immediately before they execute.

We use several expectation operations because our heuristics work with pmfs. For a task z executing on core k of multicore processor j in node i and P-state π, we compute the expected completion time (denoted $\mathrm{ECT}(i, j, k, \pi, t_l, z)$) by taking the expectation of the stochastic completion time distribution. Similarly, expected execution time (denoted $\mathrm{EET}(i, j, k, \pi, z)$) is found by taking the expectation of the execution-time distribution for a task. The expected energy consumption of an assignment (denoted $\mathrm{EEC}(i, j, k, \pi, z)$) is found by multiplying the expected execution time by the power consumption of the core and P-state where the task is assigned ($\mu(i, \pi)$), and then dividing by the power efficiency of the node where the task is assigned ($\epsilon(i)$). Note that execution time of a task is a function of P-state, as described in Sect. 6, and so completion time of a task also must be a function of P-state. Energy consumption of a task is based on the energy consumption of the core in the current P-state, as well as execution time.

Although our heuristics may manipulate pmfs and perform convolutions, their execution time is minimal relative to task execution times. The pmf operations can take a notable amount of time when there is a large number of tasks in execution queues. However, when quick scheduling decisions are needed, the queues will be mostly empty and the heuristics will operate quickly.

5.2 Shortest Queue heuristic

The *Shortest Queue* (*SQ*) heuristic [27] assigns the incoming task to the core with the fewest tasks currently assigned to it from among the feasible assignments. When invoked at time-step t_l, the heuristic finds the number of tasks currently assigned to each core k of multicore processor j in node i in the list of feasible assignments (we denote this value $|MQ(i, j, k, t_l)|$), and then maps the arriving task to the feasible assignment with the smallest value of $|MQ(i, j, k, t_l)|$. If multiple feasible assignments have the same minimum queue length, then the heuristic assigns the task to the core and P-state combination that has the minimum expected execution time for the task ($\text{EET}(i, j, k, \pi, z)$) among those with the minimum queue length.

5.3 Minimum Expected Completion Time heuristic

The *Minimum Expected Completion Time* (*MECT*) heuristic [24], assigns the incoming task to the core and P-state combination that provides the minimum expected completion time from among the feasible assignments. When invoked at time-step t_l to map task z, the heuristic finds the expectation of the completion-time distribution for task z for each feasible assignment, i.e., $\text{ECT}(i, j, k, \pi, t_l, z)$. The heuristic then maps the task to the feasible assignment with the smallest expected completion time.

5.4 Lightest Load heuristic

The *Lightest Load* (*LL*) heuristic, our new heuristic inspired by [10], defines a load quantity, and then assigns the incoming task to the core and P-state combination that has the minimum load quantity from among the feasible assignments. We define load quantity as the product of the expected energy consumption and inverse robustness, and the heuristic then tries to minimize this product. When invoked at time-step t_l to map task z, the heuristic first computes $\rho(i, j, k, \pi, t_l, z)$ and the expected energy consumption $\text{EEC}(i, j, k, \pi, z)$ for each feasible assignment. The LL heuristic then computes the load value for each potential assignment, denoted $L(i, j, k, \pi, t_l)$, as

$$L(i, j, k, \pi, t_l) = \text{EEC}(i, j, k, \pi, z) \times \big(1.0 - \rho(i, j, k, \pi, t_l, z)\big). \qquad (5)$$

The heuristic then assigns the task to the feasible assignment with the smallest load value.

5.5 Random heuristic

The *Random* (*RAND*) heuristic assigns the incoming task to a random core and P-state from among the feasible assignments. This is conceptually one of the simplest techniques for resource allocation, and we use it to contrast the benefits achieved by using the more sophisticated heuristics and our filter mechanisms.

5.6 Energy and robustness filters

We use two filtering mechanisms to restrict the set of feasible assignments a heuristic can consider. These allow us to add energy-awareness and/or robustness-awareness to a heuristic that may have neither.

Our *energy filter* restricts the set of feasible assignments by eliminating all the potential assignments that would consume more than a “fair share” of the remaining energy budget as a heuristic to try and maximize the number of tasks that will finish within the energy constraint. We denote this “fair share” estimated at time-step t_l as $\zeta_{\text{fair}}(t_l)$. A heuristic initially subtracts the energy required to run all cores in P_4 for 12 hours so that any energy wasted by idling cores is correctly accounted for. It then estimates the remaining energy as it runs by subtracting the difference between the expected energy consumption of each assignment it makes and the energy consumed by running the assigned core in P_4 for the same amount of time from the energy budget for the simulation. A heuristic utilizing start-time cancellation adds the same amount of energy back to the energy budget when a task is canceled.

If we denote the heuristic’s estimate of the remaining energy at time-step t_l as $\zeta(t_l)$, the amount of time left in the 12-hour simulation trial as t_{rem}, the total number of cores in the system as c_{total}, and the *average task execution time* (the average execution time of all tasks over all machines and all P-states) as t_{avg}, we can define a multiplier ζ_{mul} and express our “fair share” threshold as

$$\zeta_{\text{fair}}(t_l) = \big(\zeta_{\text{mul}} \times \zeta(t_l)\big) / (t_{\text{rem}} \times c_{\text{total}}) / t_{\text{avg}}. \tag{6}$$

This is the estimated energy remaining divided by an estimate of the total number of tasks that could be executed on all cores until the end of the simulation.

To deal with task-arrival bursts, we change ζ_{mul} based on the average queue depth of the system (the average of the number of tasks queued for execution or currently executing, at a single time-step). In our simulations, the best results were obtained using values of $\zeta_{\text{mul}} = 0.8$ for an average queue depth less than 0.8, $\zeta_{\text{mul}} = 1.2$ for an average queue depth of 0.8 to 1.2, and $\zeta_{\text{mul}} = 2.0$ for any average queue depth greater than 1.2.

Our *robustness filter* restricts the set of feasible assignments based on a probability threshold we denote ρ_{thresh}. The filter eliminates potential assignments of task z to core k of multicore processor j in node i and P-state π where $\rho(i, j, k, \pi, t_l, z) < \rho_{\text{thresh}}$ (i.e., potential assignments to node i, multicore processor k, core j, and P-state π which will not increase the number of expected on-time completions by at least the probability threshold). Empirically, we determined that a threshold of $\rho_{\text{thresh}} = 0.5$ worked well for limiting the set of feasible assignments without restricting a heuristic to only high-performance (and therefore high energy consumption) P-state assignments.

5.7 Start-time cancellation

As mentioned in Sect. 3.2, a heuristic can choose to cancel a task when the task is ready to start executing. When our system is oversubscribed, the buildup of tasks in each queue causes the completion-time distribution for any potential task assignment to have a broad pmf, which is the result of the large number of convolutions needed to create it. A wide completion-time pmf is generally more uncertain than a narrow one, and this extra uncertainty increases any heuristic’s probability of making a poor assignment.

By allowing a heuristic to cancel tasks exactly when they are ready to execute, the impact of uncertainty during the original task assignment can be partially mitigated.

When a task is ready to begin execution, its completion-time distribution is given by the current time convolved with the task's execution-time distribution for its assigned node and P-state. This distribution can be used to compute the probability of a task completing by its deadline with greater certainty than at assignment time, and the heuristic can then use this information to avoid spending energy executing tasks that will likely miss their deadlines anyway.

Our start-time cancellation mechanism computes the completion-time distribution of the task that is ready to begin execution and then finds the probability of it finishing by its deadline. If the task has a probability of completing by its deadline that is less than an empirically-determined constant (30% in our simulations), the task is not executed. The scheduler may cancel several tasks in a single queue before finding one that has a sufficient chance of completing by its deadline.

When start-time cancellation is used, the heuristic must anticipate the effects of cancellation when making scheduling decisions. This consists of predicting which tasks in $MQ(i, j, k, l)$ will be canceled. This is accomplished by comparing the completion-time distribution for each task in $MQ(i, j, k, l)$ with the task's deadline when calculating the stochastic completion time, robustness, or number of tasks in a queue, as described in Sect. 4. This gives an estimate of $|MQ(i, j, k, l)|$, but task-completion probabilities can change over time.

6 Simulation environment

For our simulations, we constrain our cluster configuration to limit our simulation execution times. We limit the number of multicore processors per node, $n(i)$, to values from one to four. We limit the number of cores per multicore processor, $c(i)$, to values from one to four. We also limit the total size of our cluster, N, to eight compute nodes. Finally, we assume that the efficiency of power supplies ($\epsilon(i)$) varies from at least 90% to at most 98% efficient.

A simulation trial in our environment consists of a collection of dynamically-arriving tasks. Each task's type is selected uniformly at random from one of 100 task types. We generate a distribution describing the execution time of each task type on each machine using the CVB method described in [7], with the parameters $\mu_{\text{task}} = 750$, $V_{\text{task}} = 0.25$, and $V_{\text{mach}} = 0.25$. Our entire simulation consists of four sets of 50 simulation trials, with the task arrival times, task deadlines, and task types varying across simulation trials. The task arrival rate varies between each set of 50 trials to study the effects of increasing and decreasing oversubscription during a task burst. All other parameters are held constant. Note that the simulated actual task execution times are randomly sampled from the execution time distributions during each trial, and so these vary across simulation trials.

In our simulations, we consider a bursty arrival rate [22]. We use three different arrival patterns consisting of a varying frequency of evenly-spaced task-arrival bursts with a lull of arrivals between bursts, with all bursts and lulls in a simulation having the same duration. This effectively makes the system undersubscribed between task arrival bursts, which allows heuristics and filters to try to conserve energy. Our task arrivals follow a Poisson process (as in [26]), and we define an *equilibrium rate* λ_{eq}

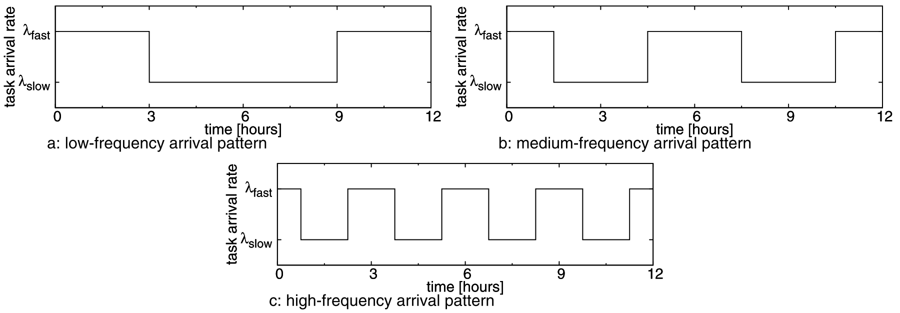


Fig. 3 The three task arrival patterns. The low-frequency arrival pattern is shown in **a**, the medium-frequency arrival pattern is shown in **b**, and the high-frequency arrival pattern is shown in **c**

such that the system is perfectly subscribed (all tasks complete by their deadlines with no energy to spare) if all tasks arrive following a Poisson process with this rate (in our simulations, $\lambda_{eq} = 1/28 \approx 0.0357$). From this parameter, we then define a *fast rate* λ_{fast} and a *slow rate* λ_{slow} such that task arrivals following a Poisson process with the fast rate will cause the system to be oversubscribed, and task arrivals following a Poisson process with the slow rate will cause the system to be undersubscribed (in our simulations, $\lambda_{fast} = 1/18 \approx 0.056$, and $\lambda_{slow} = 1/48 \approx 0.0208$). With these parameters, we generate all the task arrivals for a single simulation trial using a Poisson process where the rate is λ_{fast} for the task arrival bursts and λ_{slow} for the task arrival lulls. This varies the arrival rate such that the sum of the task-arrival bursts takes 50% of the 12-hour simulation trial, and the sum of the task-arrival lulls takes 50% of the simulation trial. The arrival rates are constant across each set of 50 simulation trials, but the arrival times may vary considerably. Figure 3 shows the three arrival patterns (low-frequency, medium-frequency, and high-frequency) used in our simulation trials.

For our simulation study, task deadlines are set for each task as the sum of its arrival time, the average execution time of its task type over all machines and all P-states, and a constant "load factor." The "load factor" represents the anticipated waiting time of a task before it begins execution. We assign deadlines assuming that each task will not have to wait longer than the t_{avg}, which we compute as the average execution time of all task types across all machines and all P-states (in our simulations, $t_{avg} \approx 1353$). We can then define the load factor as t_{avg}; the actual load will be higher when the arrival rate is λ_{fast}, and lower when the arrival rate is λ_{slow}.

The clock-speed profile for the five P-states of each core is specified by a set of five multipliers that scale the execution time distributions of a task executing on that core to reflect a higher or lower performance. The multipliers for each P-state are calculated by adding a random sample from a uniform distribution to the previous multiplier (in effect increasing the performance by a random percentage between 15% and 25%). For all cores, the minimum P-state multiplier was never less than 42% of the maximum, implying that the minimum operating frequency was at least 42% of the maximum (there are current AMD Phenom processors with similar frequency ratios [2]).

The power consumption profile for the five P-states of each core is calculated using the standard CMOS dynamic power dissipation formula (recall that power used by other system components is assumed constant and has already been accounted for in the energy constraint). If A is the number of transistor switches per clock cycle, C_L is the capacitive load, V is the supply voltage, and f is the operating frequency, then the capacitive power dissipation is

$$P_c = A \times C_L \times V^2 \times f. \tag{7}$$

We first calculate the power consumption in the highest P-state for each core by sampling a uniform random distribution between 125 and 135 watts. We then sample a uniform random distribution between 1.000 and 1.150 to get a low P-state voltage, sample a uniform random distribution between 1.400 and 1.550 to get a high P-state voltage, calculate the voltage numbers for the remaining P-states via linear interpolation, factor $A \times C_L$ into a constant, and use our voltage and frequency values for each P-state to compute a power consumption value. In practice, this results in a power consumption for the low P-state of about 25% that in the high P-state (again, there are current AMD Phenom processors with similar power consumption values [2]).

The energy constraint for our simulation ($\zeta^{\max}$) is the length of a simulation trial (12 hours) multiplied by the average power consumption over all compute nodes operating in P_2 multiplied by the total number of cores in the system. This is the amount of energy required to run the entire system for 12 hours in P_2. Because the deadlines are tight and some tasks need to execute in high-performance P-states to complete by their deadlines, this amount of energy will be insufficient to finish all tasks by their deadlines, which will force heuristics to make a trade-off.

7 Results

Box-and-whiskers plots for the results of each heuristic and its variations are shown in Figs. 4, 5, and 6. In each figure, "none," "en," "rob," and "en+rob" represent the results of the heuristic with no filtering, energy filtering only, robustness filtering only, and both energy and robustness filtering, respectively.

We first note that robustness filtering alone ("rob") has very little effect on any heuristic except RAND, regardless of the task-arrival burst frequency. This is because all of the heuristics except RAND are in some way optimizing for time. SQ assigns tasks to the shortest queue, with ties broken by the task's minimum expected execution time. MECT greedily minimizes task completion times. LL weights expected energy consumption by robustness. For these heuristics, the robustness filter rarely eliminates the assignment that would have been chosen by the heuristic without filtering. Unlike the other heuristics, RAND chooses from among the feasible assignments at random. The robustness filter limits the feasible assignments to those with at least a 30% chance of completing the task by its deadline, which gives RAND a more robust set of assignments to choose from and increases the percentage of tasks completed by their deadlines.

Although energy filtering ("en") is better than no filtering ("none") or robustness filtering ("rob") with SQ, MECT, and LL when start-time cancellation is used, it

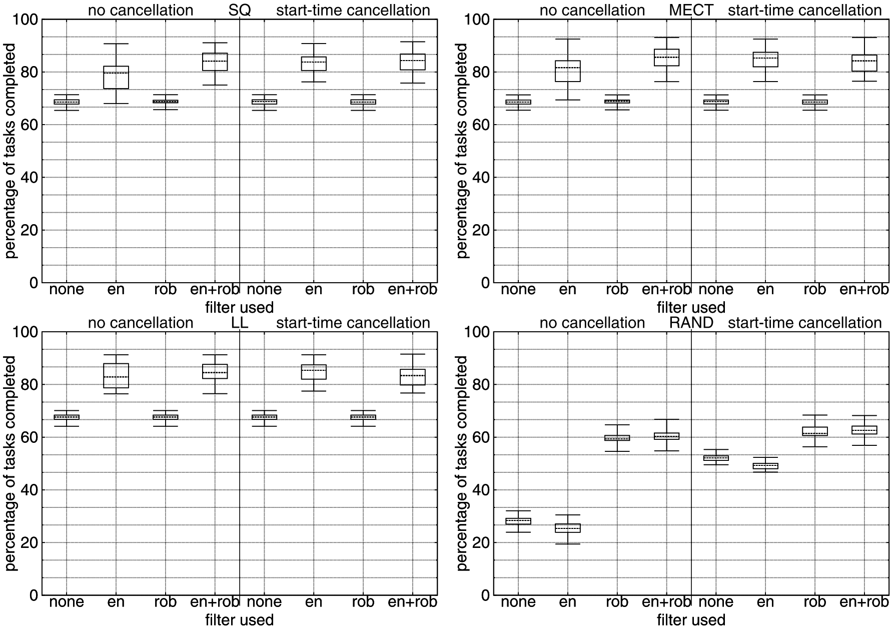


Fig. 4 The percentage of missed deadlines for all variations of all heuristics with the low-frequency arrival pattern are shown with a *box-and-whiskers* plot

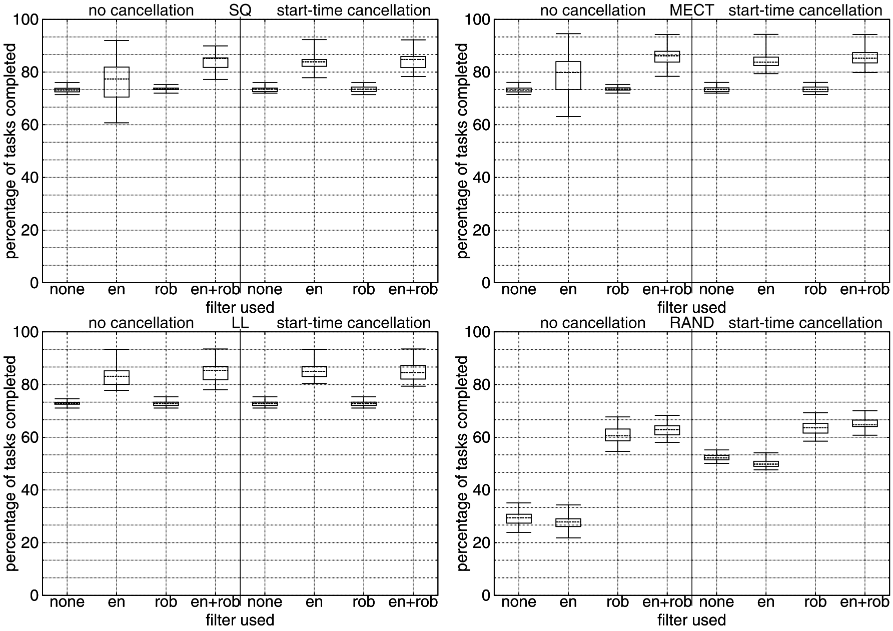


Fig. 5 The percentage of missed deadlines for all variations of all heuristics with the medium-frequency arrival pattern are shown with a *box-and-whiskers* plot

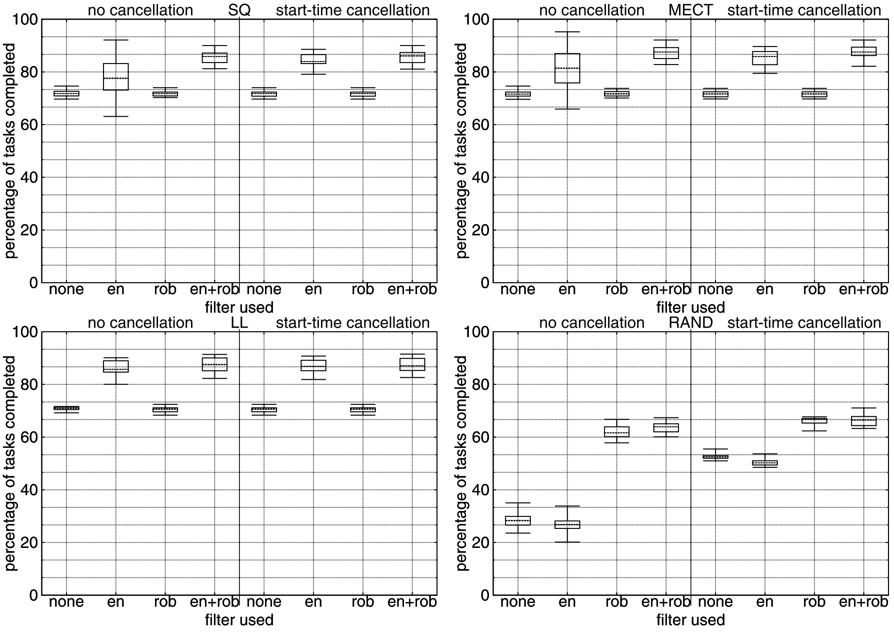
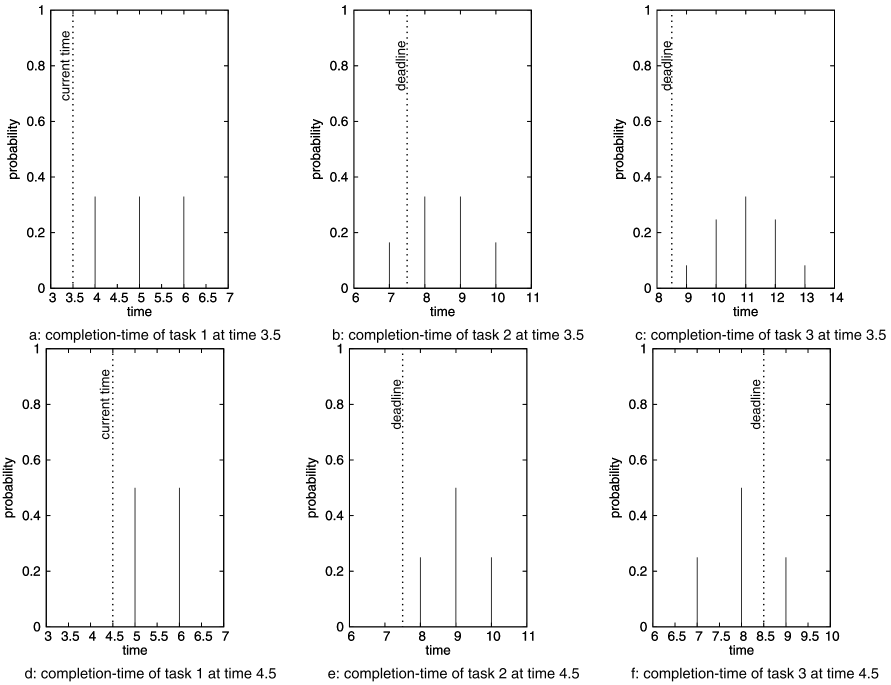


Fig. 6 The percentage of missed deadlines for all variations of all heuristics with the high-frequency arrival pattern are shown with a *box-and-whiskers* plot

is not necessarily better without start-time cancellation. This is because a heuristic using energy filtering alone will always schedule a task, even if that task has a 0% probability of completing by its deadline. Energy filtering alone cannot eliminate all potential assignments. Without start-time cancellation, this results in the execution of tasks that have little probability of completing by their deadlines, which increases the oversubscription of the system. Start-time cancellation ameliorates this problem by canceling tasks that have less than a 0.1% chance of completing by their deadlines before they start executing. We note that energy filtering alone ("en") is not better than no filtering ("none") when start-time cancellation is used for RAND because this filtering leaves RAND with only low-performance feasible assignments.

We next note that, apart from RAND and heuristics using energy filtering only, start-time cancellation does not necessarily improve the performance of a heuristic. In theory, it should always be beneficial to cancel a task that has a 0% probability of completing by its deadline. However, each heuristic must anticipate task cancellation to make appropriate resource allocation decisions: MECT to find correct expected completion times, SQ to determine correct queue depths, and LL to determine correct completion-time distributions and therefore robustness values. However, a task's probability of completion may change as previously-assigned tasks are executed and/or canceled. This means that, in practice, a heuristic may anticipate a cancellation that does not occur or fail to anticipate one that does. This results in a heuristic allocating tasks based on an incorrect assumption of the future state of the system, which can lead to poor scheduling decisions. Figure 7 shows an example of a task's probability of completion changing over time.



a: completion-time of task 1 at time 3.5

b: completion-time of task 2 at time 3.5

c: completion-time of task 3 at time 3.5

d: completion-time of task 1 at time 4.5

e: completion-time of task 2 at time 4.5

f: completion-time of task 3 at time 4.5

Fig. 7 A task's probability of completion changing over time. The completion-time for the currently executing task (task 1) at time 3.5 is shown in **a**. The completion-time for queued task 2 at time 3.5 is shown in **b**, and task 2 has a 15% chance of completing by its deadline (7.5). The completion-time for queued task 3 at time 3.5 is shown in **c**, and task 3 has a 0% chance of completing by its deadline (8.5). The completion-time for task 1 has been updated to time 4.5 in **d**. In **e**, the completion-time of task 2 at time 4.5 now indicates that task 2 has a 0% chance of completing by its deadline (7.5). In **f**, the completion-time of task 3 at time 4.5 no longer includes the execution time of task 2 (which will be canceled), and task 3 now has a 75% probability of completing by its deadline

The results indicate that arrival pattern (low-frequency, medium-frequency, or high-frequency) has little effect on heuristic performance. For instance, we notice a roughly 4% median improvement in LL with energy and robustness filtering ("en+rob") between Fig. 4 and Fig. 6. However, the intervals between the min and max values of all results decrease as the frequency of task-arrival bursts increases. This indicates that the heuristics and their filters perform more consistently across trials when there are more short task-arrival bursts. This is because the shorter periods of oversubscription cause fewer misses from poor allocation decisions.

Finally, we note that the best performance (a median of about 15% missed deadlines) was obtained by several heuristics. Note that our limited analysis of the dataset indicated that the difficulty of the workload (tight deadlines and energy constraint) caused a median of at least 7% of all missed deadlines. This indicates that an appropriate choice of filtering mechanisms is more important than the choice of heuristic. The LL, MECT, and SQ heuristics using energy and robustness filtering ("en+rob") all achieved comparable median performance on each of the three arrival patterns. With the combined energy and robustness filters, these heuristics sometimes leave tasks unassigned. By leaving tasks unassigned, the heuristics do not have to consider those task's pmfs in completion-time or queue depth computations. This reduces the uncertainty in the resource allocations, and leads to better scheduling decisions.

8 Conclusions and future work

In this paper, we developed a model of robustness and validated its use in allocation decisions. We also presented two filters, two adaptations of existing heuristics, and one new heuristic that can make assignments utilizing robustness and accounting for an energy constraint. Based on our simulation results, we can conclude that appropriate filtering mechanisms are fundamental to improving heuristic performance. Designers of resource management systems can utilize our results as modular additions to a scheduler to improve energy-efficient performance.

In future work, we would first like to perform a sort of Fourier analysis with our three arrival patterns and study the behavior of all the heuristics and filters. MECT, for instance, will perform best on a low-frequency arrival pattern if a single task arrival burst starts at the beginning of the simulation and ends at six hours. This is because MECT does not attempt to conserve energy, and will miss fewer tasks after using its energy budget. Conversely, MECT will perform poorly on a low-frequency arrival pattern with a single task-arrival burst starting at 6 hours and ending at 12 hours because it will waste energy when the system is undersubscribed instead of saving it for the arrival burst.

For other future work, we would first like to explicitly consider memory-intensive tasks in our workload and energy model. We plan to do this would be by associating a memory-intensive weighting with each task type. By multiplying using this factor when computing execution time and power consumption for different task types, we can model memory-intensive tasks that benefit less from DVFS. With the addition of memory-intensive tasks, we also will be able to consider memory energy consumption in our model. Considering a workload with tasks that do not benefit from DVFS, e.g., I/O-bound or communication-intensive tasks, would be useful. It may be possible to do this by adding a modular level on top of any existing heuristic that intercepts these task types and runs them at their "matched" P-state. We also would like to study more complex mechanisms for task cancellation. This could involve examining all tasks in a queue for cancellation, instead of just the task at the head of each queue. It is possible to derive a distribution describing a task's probability of being canceled in the current system, and such a distribution could be used to better anticipate task cancellation. We could also extend our model to include more energy-conserving techniques than just DVFS. This would include mechanisms such as ACPI G-states, power gating, and hard-disk power management. The addition of ACPI G-states and hard-disk power management will allow us to add fan and hard-disk energy consumption to our model. We also want to use full probability distributions to represent power consumption, instead of assuming that power consumption is a constant representing an average value (as described in Sect. 6). We intend to expand our model to consider tasks with varying priorities, and a system with the ability to stop tasks that miss their deadlines. Finally, we want to consider heuristics that can reassign the batch of tasks that have been scheduled but are not yet executing.

Acknowledgements The authors thank Greg Pfister and Jerry Potter for their comments on this research. A preliminary version of portions of this work appeared in [31]. We significantly expand on the work in [31] by adding a novel mechanism to cancel tasks before they start executing and by simulating with a variety of arrival patterns. More importantly, we directly deal with energy wasted by idling processors by simulating against a 12-hour window of tasks, where the research in [31] worked with a window of 1,000 tasks. The 12-hour window also makes our research more applicable to real systems.

This research was supported by the National Science Foundation under grant number CNS-0905339, and by the Colorado State University George T. Abell Endowment. This research used the CSU ISTeC HPC System supported by NSF Grant CNS-0923386.

References

1. Advanced configuration and power interface specification (2010). http://www.acpi.info/DOWNLOADS/ACPIspec40a.pdf. Accessed 2 Mar 2011
2. Advanced Micro Devices (2010) AMD Family 10h Desktop Processor Power and Thermal Data Sheet. http://support.amd.com/us/Processor_TechDocs/43375.pdf. Accessed 2 Mar 2011
3. Advanced Micro Devices (2010) AMD PowerNow! Technology. http://www.amd.com/us/products/technologies/amd-powernow-technology/Pages/amd-powernow-technology.aspx. Accessed 2 Mar 2011
4. Advanced Micro Devices (2010) BIOS and Kernel Developer's Guide (BKDG) for Family 10h Processors. http://support.amd.com/us/Processor_TechDocs/31116.pdf. Accessed 2 Mar 2011
5. Ali S, Maciejewski AA, Siegel HJ (2008) Perspectives on robust resource allocation for heterogeneous parallel systems. In: Handbook of parallel computing: models, algorithms, and applications. Chapman & Hall/CRC Press, Boca Raton, pp 41-1–41-30
6. Ali S, Maciejewski AA, Siegel HJ, Kim JK (2004) Measuring the robustness of a resource allocation. IEEE Trans Parallel Distrib Syst 15(7):630–641
7. Ali S, Siegel HJ, Maheswaran M, Hensgen D (2000) Representing task and machine heterogeneities for heterogeneous computing systems. Tamkang J Sci Eng, Special 50th Anniversary Issue 3(3):195–207.
8. Apodaca J, Young D, Briceño L, Smith J, Pasricha S, Maciejewski AA, Siegel HJ, Bahirat S, Khemka B, Ramirez A, Zou Y (2011) Stochastically robust static resource allocation for energy minimization with a makespan constraint in a heterogeneous computing environment. In: 9th ACS/IEEE international conference on computer systems and applications (AICCSA '11)
9. Aydin H, Melhem R, Mosse D, Mejia-Alvarez P (2001) Dynamic and aggressive scheduling techniques for power-aware real-time systems. In: 22nd IEEE real-time systems symposium (RTSS '01), pp 95–105
10. Barbosa J, Moreira B (2009) Dynamic job scheduling on heterogeneous clusters. In: 8th international symposium on parallel and distributed computing (ISPDC '09), pp 3–10
11. Bohrer P, Elnozahy EN, Keller T, Kistler M, Lefurgy C, McDowell C, Rajamony R (2002) The case for power management in web servers. In: Power aware computing. Kluwer Academic, Norwell, pp 261–289
12. Briceño L, Siegel HJ, Maciejewski AA, Oltikar M, Brateman J, White J, Martin J, Knapp K (2011) Heuristics for robust resource allocation of satellite weather data processing on a heterogeneous parallel system. IEEE Trans Parallel Distrib Syst 22(11):1780–1787
13. Briceño LD, Khemka B, Siegel HJ, Maciejewski AA, Groer C, Koenig G, Okonski G, Poole S (2011) Time utility functions for modeling and evaluating resource allocations in a heterogeneous computing system. In: 20th heterogeneity in computing workshop (HCW '11), pp 1–14
14. CSU Information Science and Technology Center (2011) ISTeC Cray High Performance Computing (HPC) System. http://istec.colostate.edu/istec_cray. Accessed 15 June 2011
15. Intel Corporation (2010) Frequently asked questions for Intel SpeedStep Technology. http://www.intel.com/support/processors/sb/CS-028855.htm. Accessed 2 Mar 2011
16. Iosup A, Epema D (2010) Grid workloads. IEEE Internet Comput 15(2):19–26
17. Kim JK, Siegel HJ, Maciejewski AA, Eigenmann R (2008) Dynamic resource management in energy constrained heterogeneous computing systems using voltage scaling. IEEE Trans Parallel Distrib Syst 19(11):1445–1457

18. Kim KH, Buyya R, Kim J (2007) Power aware scheduling of bag-of-tasks applications with deadline constraints on dvs-enabled clusters. In: 7th IEEE/ACM international symposium on cluster computing and the grid (CCGrid '05), pp 541–548
19. Koomey JG (2007) Estimating total power consumption by servers in the US and the world. Tech rep, Lawrence Berkeley National Laboratory. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.5562&rep=rep1&type=pdf
20. Koomey JG, Belady C, Patterson M, Santos A, Lange KD (2009) Assessing trends over time in performance, costs, and energy use for servers. Tech rep, Lawrence Berkeley National Laboratory, Stanford University, Microsoft Corporation, and Intel Corporation. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.5562&rep=rep1&type=pdf. Accessed 2 Mar 2011
21. Leon-Garcia A (1989) Probability & random processes for electrical engineering. Addison-Wesley, Reading
22. Li C, Bettati R, Zhao W (1998) Response time analysis for distributed real-time systems with bursty job arrivals. In: 1998 international conference on parallel processing (ICPP '98), pp 432–440
23. Li YA, Antonio JK, Siegel HJ, Tan M, Watson DW (1997) Determining the execution time distribution for a data parallel program in a heterogeneous computing environment. J Parallel Distrib Comput 44(1):35–52
24. Maheswaran M, Ali S, Siegel HJ, Hensgen D, Freund RF (1999) Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. J Parallel Distrib Comput 59(2):107–121
25. Phillips CL, Parr JM, Riskin EA (2003) Signals, systems, and transforms. Pearson Education, Upper Saddle River
26. Smith J, Apodaca J, Maciejewski AA, Siegel HJ (2010) Batch mode stochastic-based robust dynamic resource allocation in a heterogeneous computing system. In: 2010 international conference on parallel and distributed processing techniques and applications (PDPTA '10), pp 263–269
27. Smith J, Chong EKP, Maciejewski AA, Siegel HJ (2009) Stochastic-based robust dynamic resource allocation in a heterogeneous computing system. In: 38th international conference on parallel processing (ICPP '09)
28. Smith J, Siegel HJ, Maciejewski AA (2009) Robust resource allocation in heterogeneous parallel and distributed computing systems. In: Wah BW (ed) Wiley encyclopedia of computer science and engineering, vol 4. Wiley, Hoboken, pp 2461–2470
29. Wasserman L (2005) All of statistics: a concise course in statistical inference. Springer, New York
30. Xian C, Lu YH, Li Z (2008) Dynamic voltage scaling for multitasking real-time systems with uncertain execution time. IEEE Trans Comput-Aided Des Integr Circuits Syst 27(8):1467–1478
31. Young D, Apodaca J, Briceño L, Smith J, Pasricha S, Maciejewski AA, Siegel HJ, Bahirat S, Khemka B, Ramirez A, Zou Y (2011) Energy-constrained dynamic resource allocation in a heterogeneous computing environment. In: 4th international workshop on parallel programming models and systems software for high-end computing (P2S2 '11)
32. Yu H, Veeravalli B, Ha Y (2008) Dynamic scheduling of imprecise-computation tasks in maximizing QoS under energy constraints for embedded systems. In: 2008 Asia and South Pacific design automation conference (ASPDAC '08), pp 452–455