Makespan and Energy Robust Stochastic Static Resource Allocation of a Bag-of-Tasks to a Heterogeneous Computing System

Mark A. Oxley, Sudeep Pasricha, *Senior Member, IEEE*, Anthony A. Maciejewski, *Fellow, IEEE*, Howard Jay Siegel, *Fellow, IEEE*, Jonathan Apodaca, Dalton Young, Luis Briceño, Jay Smith, *Member, IEEE*, Shirish Bahirat, Bhavesh Khemka, Adrian Ramirez, and Yong Zou

Abstract—Today's data centers face the issue of balancing electricity use and completion times of their workloads. Rising electricity costs are forcing data center operators to either operate within an electricity budget or to reduce electricity use as much as possible while still maintaining service agreements. Energy-aware resource allocation is one technique a system administrator can employ to address both problems: optimizing the workload completion time (makespan) when given an energy budget, or to minimize energy consumption subject to service guarantees (such as adhering to deadlines). In this paper, we study the problem of energy-aware static resource allocation in an environment where a collection of independent (non-communicating) tasks ("bag-of-tasks") is assigned to a heterogeneous computing system. Computing systems often operate in environments where task execution times vary (e.g., due to cache misses or data dependent execution times). We model these execution times stochastically, using probability density functions. We want our resource allocations to be robust against these variations, where we define *energy-robustness* as the probability that the energy budget is not violated, and *makespan-robustness* as the probability a makespan deadline is not violated. We develop and analyze several heuristics for energy-aware resource allocation for both energy-constrained and deadline-constrained problems.

Index Terms—Heterogeneous computing, static resource allocation, power-aware computing, DVFS, robustness

1 Introduction

A recent study [22] estimates that the electricity used by data centers has increased by 56 percent worldwide between the years 2005 and 2010. With the electricity demands increasing and the energy costs of operating a data center surpassing 20 percent of the total costs [12], it has become common practice to either operate within an electricity budget or to reduce electricity use while maintaining service guarantees. The need for energy-efficient data centers is becoming more apparent as both power consumption and operating costs continue to rise. Energy-aware resource management techniques that can improve energy efficiency are therefore becoming increasingly important.

A commonly used technique to manage the energy efficiency in computing systems is to employ dynamic voltage and frequency scaling (*DVFS*) in server processors [14]. DVFS allows the cores in a processor to operate in discrete performance states (*P-states*), with lower-numbered P-states consuming more power but reducing the execution time of tasks. Because server processors use a large portion of the energy in a data center, we can employ DVFS to provide a trade-off between execution time and energy consumption.

Many large-scale computing facilities (e.g., data centers) incorporate heterogeneous resources that utilize a mix of different machines to execute workloads with diverse computational requirements. The execution times of tasks on heterogeneous machines are typically inconsistent such that if machine *A* is faster than machine *B* for a given task, machine *A* may not be faster for all tasks [4]. By assigning tasks to machines in an intelligent manner, it is possible to leverage machine heterogeneities to reduce task execution times and energy consumption.

The act of assigning tasks to machines is commonly referred to as resource allocation. Resource allocation decisions often rely on estimated values for task execution times whose actual values vary and may differ from available estimates (e.g., due to cache misses or data dependent execution times). These uncertainties in task execution times may cause a completion time (makespan) deadline or energy budget to be violated, therefore we want resource allocation techniques to be robust against these variations.

- *M.A. Oxley, A.A. Maciejewski, D. Young, L. Briceño, S. Bahirat, B. Khemka, A. Ramirez, and Y. Zou are with the Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO 80523. E-mail: {mark.oxley, aam, dalton.young, ldbricen, shirish.bahirat, bhavesh.khemka, adrian.ramirez, yong.zou}@colostate.edu.*
- *S. Pasricha and H.J. Siegel are with the Department of Electrical and Computer Engineering, and the Department of Computer Science, Colorado State University, Fort Collins, CO 80523. E-mail: {sudeep, hj}@colostate.edu.*
- *J. Apodaca is with the Department of Computer Science, Colorado State University, Fort Collins, CO 80523. E-mail: jonathan.apodaca@colostate.edu.*
- *J. Smith is with the Lagrange Systems, Boulder, CO 80302. E-mail: jay@lagrangesystems.com.*

Manuscript received 14 Feb. 2014; revised 4 Sept. 2014; accepted 8 Sept. 2014. Date of publication 13 Oct. 2014; date of current version 2 Sept. 2015. Recommended for acceptance by R. Kwok.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2362921

This research addresses the problem of *statically* allocating a workload of independent tasks to a heterogeneous computing cluster. Static mapping is used in several environments [4], [32], such as planning an efficient schedule for some set of jobs to be run at some time in the future. In the resource management literature, it is common to assume that information that characterizes the execution times of frequently executed tasks can be collected, e.g., [7], [20], [40], [43]. We work closely with Oak Ridge National Labs, and in their environment, as well as others, similar types of tasks are executed frequently allowing for the collection of historical information about the execution times of tasks on machines. In this study, we assume knowledge of the means and variances of the execution times of each task on each machine, and can use this information to build probability distributions that approximate historical information. We want our resource allocations to be robust against the variations in the task execution times, where we define *energy-robustness* as the probability that the energy budget is not violated, and *makespan-robustness* as the probability a makespan deadline is not violated. These probabilities are calculated using the means and variances of the task execution times.

By intelligently allocating resources and configuring DVFS, we utilize the heterogeneities in execution time and power to address two problems: (1) optimizing (maximizing) the makespan-robustness with a constraint on energy-robustness; and (2) optimizing (maximizing) the energy-robustness with a constraint on makespan-robustness. We refer to (1) as *MO-EC* (makespan-robustness optimization under an energy-robustness constraint) and (2) as *EO-MC* (energy-robustness optimization under a makespan-robustness constraint). This study focuses on the design and analysis of makespan- and energy-robust resource allocation heuristics for a heterogeneous computing cluster to address both the energy-constrained and deadline-constrained problems. We also analyze the impact of four different methods of constrained optimization used within the heuristics, and we demonstrate the flexibility and performance of the heuristics when the constraints are easy or difficult to meet. In summary, we make the following contributions:

- The design and analysis of resource management techniques for both optimizing makespan-robustness with an energy-robustness constraint (MO-EC) and optimizing energy-robustness with a makespan-robustness constraint (EO-MC).
- An enhanced power model that uses real system specifications for CPU voltage and frequency of DVFS P-states and overhead power of additional server components. Also a workload model where tasks have varying degrees of compute and memory intensity.
- An analysis on the effectiveness of our techniques on two different sized platforms that vary in both number of machines and tasks.
- A sensitivity analysis of our techniques against the level of heterogeneity.

The rest of this paper is organized as follows. Section 2 discusses related work. The system model and workload are defined in Section 3. Section 4 describes the stochastic measures for makespan and energy consumption. The energy-aware resource allocation heuristics are proposed in Section 5, and the methods of constrained optimization used in conjunction with the heuristics are discussed in Section 6. Section 7 discusses the results, and finally we give the conclusions in Section 8.

2 Related Work

Energy-aware resource allocation in high performance computing (HPC) systems is an important research area as high power consumption and associated energy costs are difficult obstacles. Therefore, numerous recent works have focused on energy-aware resource allocation techniques that exploit energy saving techniques such as DVFS to reduce the energy consumption of computing systems. To the best of our knowledge, our work is the first to address energy-aware resource allocation of a bag-of-tasks with uncertain task execution times to a heterogeneous computing system with goals of considering robustness of both makespan and energy consumption.

Energy-aware scheduling on heterogeneous platforms that considers deterministic task execution times has been previously studied (e.g., [23], [28], [29]). Li and Wu [23] consider energy-aware scheduling of a collection of independent tasks on a heterogeneous platform that is DVFS-enabled. The primary contribution is the design of a resource allocation algorithm that minimizes energy consumed while ensuring the collection of tasks completes by a common deadline. Energy-aware scheduling of a bag-of-tasks application to a heterogeneous computing system is considered in [28]. The goal in that paper is to allocate the bag of same-size tasks to the heterogeneous system to minimize energy consumed under a throughput constraint. Energy-aware scheduling on milliclusters is studied in [29]. Milliclusters are a collection of numerous low-power processing elements (e.g., those found in mobile devices) that are organized into a large cluster for scientific computing purposes. Three heuristics are designed to minimize energy consumption to complete a collection of tasks while adhering to each task's individual deadline. Our work is novel and different from those listed because it considers uncertain task execution times (modeled as random variables) and the design of relevant performance measures to capture the robustness of both makespan and energy consumption against these uncertain task execution times.

Robust resource allocation of tasks with uncertain execution times has been studied, e.g., [5], [6], [8], [18]. These works do not consider energy consumption, which is the focus of our work. Energy-aware resource allocation of tasks with uncertain execution times was studied in [42]. The goal for the resource allocation techniques in [42] is to finish as many tasks as possible by their individual deadlines without violating an energy constraint. The definition of robustness is the expected number of tasks that will complete by their individual deadlines. Our work considers complex resource management techniques designed for a static scheduling problem, probabilistic measures for both energy consumption and performance, and a more accurate power model that includes the overhead power of compute nodes and static power consumption of cores.

3 System Model

3.1 Compute Nodes

The cluster we model consists of N heterogeneous compute nodes, and each node i contains n_i homogeneous cores. Cores are the most basic processing element in this study, with each core processing one task at a time, e.g., as done in a Cray [9]. Cores are also DVFS-enabled to use P-states that allow a core to change operating voltage and frequency to provide a trade-off between the execution time and power consumption of the processor. Each core of compute node i has PS_i P-states available. We assume each core in the system can operate in an individual P-state, and our resource allocation techniques are designed such that P-states do not switch during task execution. Lower-numbered P-states consume more power, but provide faster execution times, e.g., P-state 0 provides the shortest execution times but also consumes the most power.

3.2 Power Model

The power consumption of a node includes the dynamic power of the cores, the static power consumption of the cores, and the base overhead power of the node (e.g., for disks, memory, network interface cards) [26]. We assume that when a core is finished processing its assigned workload, the core is able to deactivate and the power consumption (both dynamic and static) becomes negligible. Similarly, when all cores of a compute node are finished with their assigned workloads, the entire node is able to deactivate and consume no overhead power. For our static resource allocation problem with independent tasks, nodes do not have idle time because nodes are active when processing tasks and then deactivate when finished with their assigned workload, the ACPI S6 state [14].

For core j within node i operating in P-state π, we use the well-known equation for dynamic power ($P^{dyn}_{ij\pi}$) using the load capacitance (C_i), the supply voltage ($Vdd_{j\pi}$), and the frequency ($f_{j\pi}$) as [13]

$$P^{dyn}_{ij\pi} \propto C_i \cdot Vdd^2_{j\pi} \cdot f_{j\pi}. \tag{1}$$

We consider the static power consumption of cores and the overhead power used by additional components (e.g., memory, disks, add-on cards). We assume the static power of a core and the overhead power of a node are constant and independent of the voltage and frequency of the P-state that the core is currently in.

3.3 Workload

The workload consists of a collection of T independent tasks to be completed before a given system deadline δ and within a given energy budget of Δ. Such a collection of independent tasks is known as a *bag-of-tasks*. We assume the tasks to be executed are known *a priori* such that we can perform a static (i.e., off-line) mapping. The actual values of task execution times vary (e.g., due to cache misses or data dependent execution times), and we model the execution times stochastically to account for these variations. That is, we are provided a mean and a variance that describe an execution time probability density function (*pdf*) for each task executing on each compute node in each P-state. In an actual system, the means and variances of these Gaussian execution time distributions can be approximated using historical, experimental, or analytical techniques [17], [24], [34]; in Section 7 we discuss our values for evaluation. The use of Gaussian distributions allows us to sum task execution times using a closed-form equation rather than having to perform convolution, however our proposed resource management techniques are applicable for task execution times that are described by any distribution.

It has been shown [21] that the arithmetic intensity (the number of operations performed per word of memory transferred) of a workload can greatly affect how the execution time of a task scales with the frequency of the CPU. The execution times of tasks with high arithmetic intensity generally scale proportionally with frequency, e.g., when the frequency is halved the execution time of the task doubles. Because the overhead power of a compute node remains constant regardless of P-state, the greater execution times resulting from low-power P-states can result in greater energy consumption than when executing the task as fast as possible in P-state 0. The most energy-efficient P-state depends on the ratio of compute energy saved by operating in a low-power P-state to the amount of overhead energy consumed over the longer execution time that results from the decrease in frequency. With memory intensive workloads (i.e., workloads with low arithmetic intensity), a reduction in frequency has less impact on the execution times of tasks, because the processor spends a large portion of cycles waiting for data from memory. Memory intensive tasks offer greater potential for energy savings using DVFS, as power can be greatly reduced with little effect on execution time. We assume each task is one of a set of *task types* that is representative of the arithmetic intensity of the task. Our method for the scaling of execution time by task type and frequency is detailed in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2014.2362921.

4 Stochastic Measures

4.1 Overview

The traditional performance measure for bag-of-tasks scheduling problems is *makespan*, the time required for all tasks within the bag to finish execution. In this study, we also are concerned with the energy consumption required to execute the bag-of-tasks, however, typically both makespan and energy are measures that rely on deterministic values for time. Because real environments have uncertainty, resource allocation decisions in such environments should be based on this stochastic information and be robust against the variations in the task execution times.

We define a "robust" resource allocation as one that can mitigate the impact of uncertainties on both performance and energy objectives. To claim robustness for a system, the following three questions must be answered [2]: (1) *What behavior makes the system robust?* In our system, a resource allocation is considered *makespan-robust* if the entire workload completes within the system deadline δ, and *energy-robust* if the workload can be completed within an energy budget of Δ. (2) *What uncertainties is the system robust against?*

We want our system to be robust against the uncertain task execution times. (3) *How is the robustness of the system quantified?* We quantify the makespan-robustness of a resource allocation as the probability that the entire workload is completed by δ, and the energy-robustness as the probability that the workload uses no more than Δ energy.

4.2 Makespan-Robustness

The execution time distribution for each task on each node is modeled as a Gaussian distribution with a given mean and variance. For a given resource allocation, let the set T_{ij} denote tasks in T that have been assigned to core j in compute node i and let $t_{ij}^x \in T_{ij}$ where $1 \leq x \leq |T_{ij}|$. Let $\mathbf{PS}(t_{ij}^x)$ denote the assigned P-state for task t_{ij}^x. We denote the mean execution time associated with task t_{ij}^x executed in P-state π as $\mu(t_{ij}^x, \pi)$ and the associated variance as $V(t_{ij}^x, \pi)$.

The calculation of the completion time of core j when using a stochastic model for task execution times is performed by taking the convolution of the random variables (representing the execution times of tasks) for all tasks assigned to core j. The convolution of two independent normally distributed random variables α and β produces a normally distributed random variable with its mean being the sum of the means of α and β and the variance being the sum of the variances of α and β.

The expected finishing time of core j in compute node i, denoted F_{ij}, is the sum of the mean execution times of all tasks assigned that core and is given as

$$F_{ij} = \sum_{\forall t_{ij}^x \in T_{ij}} \mu\big(t_{ij}^x, \mathrm{PS}(t_{ij}^x)\big). \tag{2}$$

The variance of the completion time distribution of core j, denoted σ_{ij}^2, is the sum of the variances of the execution times of all tasks assigned to that core and is given as

$$\sigma_{ij}^2 = \sum_{\forall t_{ij}^x \in T_{ij}^x} V\big(t_{ij}^x, \mathrm{PS}(t_{ij}^x)\big). \tag{3}$$

Thus, the completion time distribution of all tasks assigned to core j in compute node i can be expressed as the distribution $\mathcal{N}(F_{ij}, \sigma_{ij}^2)$. Given deadline δ, we can compute the probability that $\mathcal{N}(F_{ij}, \sigma_{ij}^2)$ is less than δ by converting $\mathcal{N}(F_{ij}, \sigma_{ij}^2)$ to its associated cumulative density function (*cdf*) and finding the probability associated with that core finishing before time δ (i.e., $\mathbb{P}(\mathcal{N}(F_{ij}, \sigma_{ij}^2) \leq \delta)$). We define the overall *makespan-robustness* of a resource allocation, denoted Ψ, as the minimum probability across all cores. That is, each core has a probability of at least Ψ that it will complete its assigned workload by deadline δ. We calculate Ψ as

$$\Psi = \min_{\forall i \in N} \Big(\min_{\forall j \in n_i} \mathbb{P}(\mathcal{N}(F_{ij}, \sigma_{ij}^2) \leq \delta)\Big). \tag{4}$$

We denote the makespan-robustness constraint as Γ, so for EO-MC we require resource allocations to meet this constraint (i.e., $\Psi \geq \Gamma$). When using a distribution other than the normal distribution for task execution times, the completion time for all tasks on a given core can be calculated by taking the convolution of all of the task execution time random variables assigned to that core. The makespan-robustness can then be calculated by converting the result into its associated cdf and then finding the probability associated with that core finishing before the deadline.

4.3 Energy-Robustness

The energy required to process the workload is determined by summing the energy used by all tasks in the workload. In our model, the overhead power of nodes (O_i), the dynamic power of cores ($P_{ij\pi}^{dyn}$), and the static power of cores ($P_{ij\pi}^{stat}$) all contribute to the total energy consumed. These power values are multiplied by the mean execution times of tasks to calculate the expected energy to process the workload, or by the variances of the execution times of tasks to calculate the variance in energy consumed to process the workload. Let T_{ij}^π denote the subset of tasks assigned to core j of compute node i processed in P-state π, i.e., $T_{ij}^\pi = \{\forall t_{ij}^x \in T_{ij} \mid PS(t_{ij}^x) = \pi\}$. The energy consumed is calculated as the product of execution time (a random variable) and average power (a deterministic value). The multiplication of a random variable with a scalar value has the effect of multiplying the expected value by that value and the variance by the square of that value [15]. The expected dynamic energy spent by core j in compute node i at P-state π, denoted $Mean_{ij\pi}^{dyn}$, is the sum of the mean values of dynamic energy consumption for all tasks assigned to that core and is given as

$$Mean_{ij\pi}^{dyn} = \sum_{t_{ij}^x \in T_{ij}^\pi} P_{ij\pi}^{dyn} \cdot \mu\big(t_{ij}^x, \pi\big). \tag{5}$$

Likewise, the variance of the dynamic energy spent by core j in compute node i in P-state π, denoted $Var_{ij\pi}^{dyn}$, is the sum of the variance values for dynamic energy consumption for all tasks assigned to that core and is given as

$$Var_{ij\pi}^{dyn} = \sum_{t_{ij}^x \in T_{ij}^\pi} \big(P_{ij\pi}^{dyn}\big)^2 \cdot V\big(t_{ij}^x, \pi\big). \tag{6}$$

Similarly, the expected static energy consumed by core j on node i, denoted $Mean_{ij\pi}^{stat}$, is the sum of the mean values of static energy consumption for all tasks assigned to that core and is given as

$$Mean_{ij\pi}^{stat} = \sum_{t_{ij}^x \in T_{ij}^\pi} P_i^{stat} \cdot \mu\big(t_{ij}^x, \pi\big). \tag{7}$$

The variance of static energy, denoted $Var_{ij\pi}^{stat}$, is the sum of the variance values of static energy consumption for all tasks assigned to that core and is given as

$$Var_{ij\pi}^{stat} = \sum_{t_{ij}^x \in T_{ij}^\pi} \big(P_i^{stat}\big)^2 \cdot V\big(t_{ij}^x, \pi\big). \tag{8}$$

Recall that a node remains active and consumes overhead power (O_i) until all cores within the node are finished with their workload. Let F_i be the maximum expected completion time among cores in node i, and σ_i^2 be the associated variance. The energy required to process the workload includes the overhead power, the dynamic power consumed by cores, and the static energy consumed by cores.

We calculate the expected energy required to process the entire workload across all compute nodes, denoted ζ, as

$$\zeta = \sum_{i=1}^{N}\left(F_i \cdot O_i + \sum_{j=1}^{n_i}\sum_{\forall \pi \in PS_i}\left(Mean_{ij\pi}^{dyn} + Mean_{ij\pi}^{stat}\right)\right). \quad (9)$$

The variance of the energy required to process the entire workload, denoted γ, is

$$\gamma = \sum_{i=1}^{N}\left(\sigma_{ij}^2 \cdot (O_i)^2 + \sum_{j=1}^{n_i}\sum_{\forall \pi \in PS_i}\left(Var_{ij\pi}^{dyn} + Var_{ij\pi}^{stat}\right)\right). \quad (10)$$

The distribution for the total energy consumed to process the workload can be expressed as $\mathcal{N}(\zeta, \gamma)$. Given an energy budget of Δ, we can compute the probability that $\mathcal{N}(\zeta, \gamma)$ is less than Δ by converting $\mathcal{N}(\zeta, \gamma)$ to its associated cdf and finding the probability that the energy required to process the workload is less than Δ (i.e., $\mathbb{P}(\mathcal{N}(\zeta, \gamma) \leq \Delta)$). We denote this probability as ϕ, i.e., ϕ is the *energy-robustness* of a resource allocation. The energy-robustness constraint is denoted η, so for MO-EC we require resource allocations to meet this constraint (i.e., $\phi \geq \eta$).

5 Heuristics

5.1 Overview

The goal of this study is to design and analyze resource allocation heuristics with two different goals, MO-EC and EO-MC. In this section, we present three greedy heuristics and three non-greedy heuristics that have been adapted for our environment. The minimum expected energy (Min-Energy) and Min-Min Completion Time (*Min-Min CT*) heuristics provided poor results, but were found to be useful as seeds for our non-greedy heuristics. We define a solution generated by a resource allocation technique as a complete mapping of tasks to both cores and P-states. Though we use Gaussian distributions for task execution times in our simulation study, our heuristics can use the mean values of any distribution to perform resource allocation.

5.2 Minimum Expected Energy

The minimum expected energy heuristic greedily assigns each task to the maximum makespan-robustness core in the node and P-state combination that minimizes the *expected* energy consumption of the task.

5.3 Min-Min Completion Time

Min-Min Completion Time is a two-phase greedy heuristic, based on concepts in [4], [16], [25], [37]. We consider two variations of the heuristic, Min-Min P_{max} and Min-Min P_{min}, that differ in how P-state assignments are selected.

Min-Min P_{min}. All tasks are initially "unmapped" (placed in the *unmapped batch*). Each unmapped task is then paired to the core that yields the *m*inimum *e*xpected *c*ompletion *t*ime (*MECT*) when each core is considered to be executing in the *lowest-numbered* P-state (P-state 0). In the second phase, the task-core combination that yields the overall MECT is selected for assignment in P-state 0, and the task is removed from the unmapped batch. The ready times of all cores are updated and the heuristic begins another iteration. This process continues until all tasks are mapped (i.e., the unmapped batch is empty).

Min-Min P_{max}. Same as Min-Min P_{min} except using the highest-numbered P-state.

5.4 Min-Min Balance

Min-Min Balance starts from an initial solution (either Min-Min P_{min} or Min-Min P_{max} and tries to improve the solution using greedy modifications. The *minimum makespan-robustness core*, denoted $core_{minM}$, refers to the core that has the least probability of finishing its assigned workload by the deadline, that is, the core that determines the makespan-robustness measure of the solution. The *maximum makespan-robustness core*, denoted $core_{maxM}$, refers to the core that has the greatest probability of finishing its workload by the deadline. We now show how we design the Min-Min Balance heuristic for MO-EC and EO-MC.

MO-EC. We start by generating an initial mapping using Min-Min P_{max}. The solution is then modified to increase makespan-robustness by reassigning tasks and P-states, keeping moves that improve the solution (i.e., greater makespan-robustness value without violating the energy-robustness constraint). The first step reassigns an arbitrary task from $core_{minM}$ to $core_{maxM}$ in the lowest-numbered P-state that does not violate the energy-robustness constraint. Then $core_{minM}$ and $core_{maxM}$ are recalculated, and this step is repeated until any task transferred from $core_{minM}$ does not result in an improved solution. The second step changes the P-state of an arbitrary task on $core_{minM}$ to a lower-numbered (i.e., better performing) P-state unless the energy-robustness constraint will be violated. If the constraint is not violated, $core_{minM}$ is recalulated and the process is repeated until decreasing the assigned P-state of any task on $core_{minM}$ violates the energy-robustness constraint.

EO-MC. We start by generating an initial mapping using Min-Min P_{min}. We use two steps that modify the allocation to improve energy-robustness, keeping moves that improve the solution (i.e., better energy-robustness without violating the makespan-robustness constraint). The first step reassigns an arbitrary task from $core_{minM}$ to $core_{maxM}$ in the P-state that most improves energy-robustness without violating the makespan-robustness constraint. Then $core_{minM}$ and $core_{maxM}$ are recalculated, and this step is repeated until any task transferred from $core_{minM}$ does not result in an improved solution. The second step increases the value of the assigned P-state of an arbitary task on $core_{maxM}$ by one unless the makespan-robustness constraint will be violated. If the constraint is not violated, $core_{maxM}$ is recalculated and the process is repeated until increasing the assigned P-state of any task on $core_{maxM}$ violates the constraint.

5.5 Tabu Search

5.5.1 Overview

The distinguishing feature of Tabu Search is its exploitation of memory through the use of a *Tabu List* [11]. We use a Tabu List to store regions of the search space that have been searched and should not be searched again. Our implementation of Tabu Search, based on concepts in [4], combines intelligent local search ("short hops") with global search ("long-hops") in an attempt to find a globally optimal solution.

Local search is performed using three short-hop operators: (1) *task swap*, (2) *task reassignment*, and (3) *P-state reassignment*. One short-hop consists of one iteration of all three operators. Long-hops are performed when local search terminates, with the purpose of jumping to a new neighborhood in the search space, while avoiding areas already searched. After each long-hop, short-hops are again performed to locally search the region near the long-hop solution. The Tabu List stores unmodified long-hops (i.e., starting solutions) that indicate neighborhoods that have been searched before, and may not be searched again. A new solution generated by a long-hop must differ from any solution in the Tabu List by 25 percent of the task-to-core and P-state assignments, otherwise a new long-hop solution is generated. Pseudo-code for the Tabu Search heuristic is given in Algorithm 1.

Algorithm 1. Pseudo-Code for Our Tabu Search Heuristic

```
1. while termination criteria not met do
2.   generate new long-hop, avoiding Tabu areas
3.   while solution is improving do
4.     task swap
5.     task reassignment
6.     P-state reassignment
7.   end while
8. end while
```

We now discuss how long-hops are performed and the purpose of the mean rank matrix before detailing the three short-hop operators (*task swap*, *task reassignment*, and *P-state reassignment*).

5.5.2 Long-Hops

The purpose of a long-hop is to jump to new areas of the solution space to begin a new local search (i.e., short-hops), while avoiding areas of the search space that have already been searched through the use of a Tabu List. The initial solution (first long-hop) is generated using the appropriate Min-Min Balance allocation (for MO-EC or EO-MC) to help ensure that the constraints are met. Subsequent long-hop solutions are generated by first unmapping 25 percent of arbitrary tasks then reassigning them using Min-Min Balance, as before.

5.5.3 Mean Rank Matrix

We introduce the concept of a mean rank matrix that contains the rank of each heterogeneous node for each task, based on mean execution times. That is, for a given task, the nodes are ranked by how fast the nodes can execute the task (e.g., if node i can execute task t faster than node j, node i is given a better rank for task t). Let the rank of task t on node i be $rank(t, i)$, where $1 \leq rank(t, i) \leq N$. We define the best-ranked (fastest) node for a task as the *rank 1 node*. When comparing the rank of any two tasks A and B on node i, task A is ranked lower (better) than task B if $\text{rank}(A, i)$ is less than $\text{rank}(B, i)$. The mean rank matrix is used in some of the short-hop operators.

5.5.4 Short-Hops Overview

The short-hop operators are used to perform greedy local search on a solution generated by a long-hop. *Task swap* swaps two tasks between two different cores, *task reassignment* transfers a task from one core to another, and *P-state reassignment* changes the P-state of a task. The *task reassignment* and *P-state reassignment* operators modify the assignments of specific tasks and cores, whereas *task swap* incorporates some randomness by selecting arbitrary cores to swap tasks. We found that incorporating some randomness with greedy intelligence provided the best results. We tried several variations of the three short-hop operators but only present our best-performing methods for the sake of brevity. The decisions made by the three short-hop operators change depending on whether it is desired to solve MO-EC or EO-MC. We first detail the operators when designed for MO-EC, then explain changes when designed for EO-MC.

5.5.5 Short-Hop Operators for MO-EC

Task swap. The goal of the *task swap* operator is to swap tasks that are assigned to poorly (high) ranked nodes to better (low) ranked nodes, a move that can potentially improve both makespan and energy-robustness. We divide the task swap operator into four steps. (1) We first choose an arbitrary core j and create a task list consisting of all tasks assigned to core j on compute node i, recalling that the notation for such a task list is T_{ij}. (2) T_{ij} is sorted in descending order by the rank of each task for node i. (3) We select the first task in the list, denoted $\boldsymbol{task_A}$, and find the rank 1 node for the task, denoted $\boldsymbol{node_{best}}$. Within $\boldsymbol{node_{best}}$, an arbitrary core z is chosen. The task from core z that has the lowest rank for node i, denoted $\boldsymbol{task_B}$, is selected for swap. (4) The core assignments for $\boldsymbol{task_A}$ and $\boldsymbol{task_B}$ are swapped, and the best P-state combination (according to the method of constrained optimization used) is found to run the cores in when executing the tasks. If the solution improves, the swap is kept and *task swap* ends. Otherwise, the swap is not kept, and *task swap* repeats using the next task in the list T_{ij} until the solution improves or all tasks in T_{ij} have been considered.

Task reassignment. The goal of *task reassignment* is to improve makespan-robustness by transferring tasks from the core with the worst makespan-robustness to another core. Task reassignment consists of three steps. (1) Find the minimum makespan-robustness core (core j on compute node i) and create a task list consisting of all tasks assigned to that core (T_{ij}). (2) T_{ij} is sorted in descending order by the rank of each task for node i. (3) We select the first task in the list ($\boldsymbol{task_A}$), and find the rank 1 node for the task ($\boldsymbol{node_{best}}$). Within $\boldsymbol{node_{best}}$, the highest makespan-robustness core is selected as the target core (core z), and $\boldsymbol{task_A}$ is assigned to core z in the best P-state (according to the constrained optimization method). If the solution improves, the new assignment is kept and *task reassignment* ends. Otherwise, the new assignment is not kept, and *task reassignment* repeats using the next task in the list T_{ij} until the solution improves or all tasks in T_{ij} have been considered.

P-state reassignment. The goal of *P-state reassignment* is to change P-state assignments of tasks to greedily optimize the performance metric if the constraint is met, or to greedily meet the constraint if the constraint is violated. If the energy-robustness constraint has been met (i.e., $\phi \geq \eta$), the minimum makespan-robustness core (core *j* on compute node *i*) is chosen, and a task list is generated consisting of all tasks assigned to that core (T_{ij}). A task is chosen arbitrarily from the list ($task_A$), and the P-state of the task is decreased by 1 if not already currently assigned to execute in P0.

If the energy constraint has not been met (i.e., $\phi < \eta$), the maximum makespan-robustness core (core *j* on compute node *i*) is chosen, and a task list is generated consisting of all tasks assigned to that core (T_{ij}). A task is chosen arbitrarily from the task list ($task_A$), and the P-state of the task is changed to the one that gives the highest system-wide energy-robustness.

For both cases, if the solution improves, the new P-state is kept and *P-state reassignment* ends. Otherwise, the new P-state is not kept, and *P-state reassignment* repeats using the next task in the list T_{ij} until the solution improves or all tasks in T_{ij} have been considered.

5.5.6 Short-Hop Operators for EO-MC

Task swap. We make the following two changes to *task swap* to optimize for EO-MC instead of MO-EC. First, step 2 is changed to sort T_{ij} in descending order of expected energy consumption instead of rank. Second, Step 3 is changed to find the minimum energy node for $task_A$ and selects $task_B$ from core *z* that consumes the least expected energy for node *i*.

Task reassignment. We change step 2 of *task reassignment* from MO-EC to sort T_{ij} in descending order by expected energy consumption instead of rank, and in step 3 the minimum energy node is found as the destination for the transfer of $task_A$ rather than the rank 1 node.

P-state reassignment. If the makespan-robustness constraint has been met (i.e., $\Psi \geq \Gamma$), the maximum makespan-robustness core (core *j* on compute node *i*) is chosen, and a task list is generated consisting of all tasks assigned to that core (T_{ij}). A task is chosen arbitrarily from the list ($task_A$), and the P-state of the task is assigned to the P-state that gives the best energy-robustness.

If the makespan-robustness constraint has not been met (i.e., $\Psi < \Gamma$), the minimum robustness core (core *j* on compute node *i*) is chosen, and a task list is generated consisting of all tasks assigned to that core (T_{ij}). A task is chosen arbitrarily from the list ($task_A$), and the P-state of the task is decreased by 1 if not already currently assigned to execute in P0.

5.6 Genetic Algorithm

5.6.1 Overview

Genetic algorithms have been shown to be effective in resource allocation and job shop scheduling problems (e.g., [4], [10]). The Genitor [38] GA implemented in this study operates on a population of 200 chromosomes (determined empirically). Each chromosome is used to represent a solution (i.e., a complete resource allocation). A chromosome consists of a collection of $|T|$ genes, where each gene represents a task assignment to a core and P-state. The initial population is generated using four solutions generated heuristically using Min-Energy, Min-Min P_{max} and P_{min}, and Min-Min Balance heuristics (using the appropriate Min-Min Balance method for MO-EC or EO-MC), and 196 randomly generated solutions.

After the initial population generation, all chromosomes in the population are evaluated and ranked (based on the method of constrained optimization used, detailed in Section 6). *Crossover* and *mutation* operators are used to generate offspring chromosomes by altering existing solutions. The GA enforces the population size by eliminating the lowest-ranked chromosomes such that the population remains fixed at its original size. We now discuss how crossover and mutation are used to generate new offspring.

5.6.2 Crossover and Mutation

Two crossover operators are used to swap task assignments or P-states between chromosomes: i) *task-assignment crossover* and ii) *P-state crossover*. Both crossover operators start by selecting two parent chromosomes using a linear bias [38] and two crossover points *x* and *y* are generated such that $x < y \leq |T|$. In *task-assignment crossover*, all of the task-to-core assignments in genes ranging from *x* to *y* of the first chromosome are swapped with all of the task-to-core assignments in genes ranging from *x* to *y* of the second chromosome. Because cores on different nodes may have different numbers of P-states available, if a task changes nodes we assign the task in the P-state that is closest to the clock frequency of its previous assignment. After *task-assignment crossover*, *P-state crossover* is performed. *P-state crossover* also considers the offspring generated by task-assignment crossover when selecting parent chromosomes (i.e., an intermediate population of 202 chromosomes). In *P-state crossover*, all of the P-state assignments in genes ranging from *x* to *y* of the first chromosome are swapped with all of the P-states in genes ranging from *x* to *y* of the second chromosome. Assume an offspring is created from genes 1 to $x - 1$ and $y + 1$ to $|T|$ of parent *A*, and genes *x* to *y* of parent B. Gene *i* ($x \leq i \leq y$) of the offspring is assigned to the P-state of parent *B* that most-closely matches the frequency of the P-state of gene *i* in parent *A*. The crossover operators generate two new offspring each (four total).

Two mutation operators are used to alter task-to-core assignments and P-states: i) *task-assignment mutation*, and ii) *P-state mutation*. *Task-assignment mutation* is probabilistically performed on both of the offspring generated by *task-assignment crossover* and *P-state mutation* is probabilistically performed on both of the offspring generated by *P-state crossover*. For both mutations, offspring chromosomes have a probability p_m of being mutated (empirically set to 0.1). If a chromosome is selected for mutation, each gene in that chromosome has a probability p_{mg} of being mutated (empirically set to 0.001). In *task-assignment mutation*, if a gene is mutated the task corresponding to that gene is assigned to a random core (in a P-state selected as in *task-assignment crossover*). In *P-state mutation*, if a gene is mutated the task corresponding to that gene is assigned to a random P-state. After crossover and mutation, offspring chromosomes are added to the population, evaluated (as specified in Section 6), and the least-fit chromosomes are discarded to bring the population back to its original size. This completes one generation of the GA. This process is repeated for a predetermined time limit (see Section 7), and the most-fit chromosome is returned as the solution.

5.6.3 Differences between MO-EC and EO-MC

The genetic algorithm is versatile because its intelligence lying in how solutions are evaluated and ranked. Therefore the only differences between MO-EC and EO-MC are the different Min-Min Balance seeds used and how the chromosomes are ranked (detailed in Section 6).

5.7 Genetic Algorithm with Local Search (GALS)

Our genetic algorithm with local search combines the population-based global search from the GA with the local search techniques from our Tabu Search heuristic. After both crossover and mutation operations are finished (as performed in the GA), a local search is applied to the offspring chromosomes using the three local search operators from our Tabu Search heuristic under the condition that an offspring chromosome is *at least* as "good" as the worst chromosome in the population, so as to not waste time trying to improve a poor solution. The number of iterations of local search on each offspring chromosome was experimentally found to provide the best results at 200 iterations.

The differences between MO-EC and EO-MC for GALS are the same as for the GA, and the short-hop operators from Tabu Search (detailed in Sections 5.5 and 5.6).

6 Constrained Optimization

6.1 Overview

The previous section described the heuristics we propose to use for our resource allocation problem. Incorporating constraints into heuristics that are typically designed to optimize for an unconstrained objective (e.g., Tabu Search and GA) is a difficult problem and a research topic in itself. In this section we present several constraint-handling methods adapted from the literature [31], [33], [35], [41] that we use in combination with some of the proposed heuristics. These methods help us determine solutions that are "better" than other alternatives over the search space examined by the heuristics. In this section, we present the "static penalty function," "dynamic penalty function," and "superiority of feasible solutions" techniques. Details of the less-effective "limiting the search space" technique can be found in Appendix C, available in the online supplemental material.

6.2 Static Penalty Function

The static penalty function technique [31], [35], [41] reduces the objective function value of infeasible solutions based on the solution's distance from feasibility, but still allows infeasible solutions to be considered when searching for an optimal feasible solution. The distance from feasibility of a solution for MO-EC is

$$d_\phi = \eta - \phi. \tag{11}$$

For EO-MC, the distance from feasibility is

$$d_{\boldsymbol{\Psi}} = \boldsymbol{\Gamma} - \boldsymbol{\Psi}. \tag{12}$$

We denote c as a constant used to control how strongly a constraint will be enforced. Our penalized objective function for MO-EC is

$$\psi_\Psi = \begin{cases} \boldsymbol{\Psi} - c \cdot d_\phi & \text{if } d_\phi > 0 \\ \boldsymbol{\Psi} & \text{if } d_\phi \leq 0. \end{cases} \tag{13}$$

Our penalized objective function for EO-MC is

$$\psi_\phi = \begin{cases} \phi - c \cdot d_{\boldsymbol{\Psi}} & \text{if } d_{\boldsymbol{\Psi}} > 0 \\ \phi & \text{if } d_{\boldsymbol{\Psi}} \leq 0. \end{cases} \tag{14}$$

When d_ϕ or d_Ψ are greater than zero, it indicates that the constraint has not been met. We then penalize the objective functions (ψ_ϕ or ψ_Ψ) by subtracting a weighted value of the distance from feasibility. A high value of the coefficient c (i.e., high penalty for an infeasible solution) can produce low quality solutions by restricting exploration of the infeasible region. However, c must be large enough that a feasible solution is found. In our experiments, the best results were obtained when setting c to 2.

When d_ϕ or d_Ψ are less than or equal to zero it indicates the constraint has been met. Solutions are not rewarded when d_ϕ or d_Ψ are less than zero, as all that matters is the constraint is not violated. Heuristics incorporating the static penalty function return the best solution encountered that meets the constraint.

6.3 Dynamic Penalty Function

The static penalty function has a primary deficiency in that solutions obtained greatly depend on the penalty weight c, and a "good" value for c will vary depending on the heuristic used and even from iteration to iteration within a heuristic. A dynamic penalty function [35], [41] uses knowledge of the current search state to guide the search along the boundary of feasibility where the optimal solution is likely to occur. For evolutionary algorithms (e.g., GA and GALS), this is done by adjusting the penalty weight to guide the search in such a way that the population has an equal number of feasible and infeasible solutions. We set the penalty weight (c) to 2, and at the end of each generation of the GA or GALS, the penalty weight is increased by a small amount (0.01) if less than half of the population are feasible solutions or decreased by a small amount (0.01) if at least half of the population are feasible solutions. For Tabu Search, the penalty weight is increased by 0.01 at the end of an iteration if the solution is infeasible or decreased by 0.01 if feasible. Heuristics incorporating the dynamic penalty function return the best solution encountered that meets the constraint.

6.4 Superiority of Feasible Solutions

By adopting the rule that any feasible solution is better than any infeasible solution [33], it is possible to avoid experimentally tuning penalty parameters. Our heuristics use this method in the following way: (1) any feasible solution is better than any infeasible solution, (2) when two infeasible solutions are compared, the one with the smallest distance from feasibility (see Equations (11) and (12)) is considered better, and (3) when two feasible solutions are compared, the one with the better objective function value is considered better.

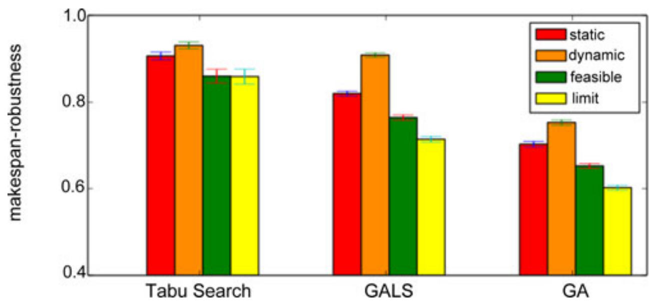


Fig. 1. Comparison of constrained optimization techniques for MO-EC with Tabu Search, GALS, and GA (25 node system, 458 total cores, and 10,000 tasks). The system deadline was set to 15,500 seconds, the energy budget was set to 58 MJ, and the energy-robustness constraint was set to 90 percent.

7 Results

We consider two different simulation sizes in our simulation study. The small simulation size consists of 25 compute nodes (N), based on 25 different servers listed in the SPECpower_ssj2008 results [36], and 10,000 tasks. The large simulation size consists of 250 compute nodes and 100,000 tasks. In Appendix A, available in the online supplemental material, we provide data collected from SPECpower_ssj2008, give details on how we use this data for our system parameters, provide information on how we obtain our voltage/frequency values for P-states, and workload generation details.

We conducted simulations to find a balance of short-hops and long-hops in Tabu Search (see Appendix B, available in the online supplemental material), compare the different heuristics using our constrained optimization techniques (Fig. 1), demonstrate the effectiveness of each heuristic at handling various degrees of difficulty to meet the deadline and energy budget (Figs. 2 and 3), analyze the trends on the large simulation size compared to the small simulation size (Figs. 4 and 5), and perform a sensitivity analysis of our heuristics across environments of varying heterogeneity (Fig. 6). Results in this section show the mean and 95 percent confidence interval error bars of 96 trials, with the means and variances for task execution times varying between trials. These trials simulate numerous diverse heterogeneous workload/system environments. For the sake of brevity, we do not include results for the Min-Energy or Min-Min P_{min} and P_{max} heuristics as they performed poorly by themselves but were found useful as seeds in the GA, GALS, and Tabu Search heuristics. Unless otherwise stated, for experiments considering the small simulation size, the system deadline (δ) was set to 15,500 seconds, the energy budget (Δ) was set to 58 megajoules (MJ). For the large simulation size, the system deadline was set to 13,500 seconds, the energy budget was set to 580 MJ. In both cases, the energy/makespan-robustness constraints (η and Γ) were set to 90 percent.

Fig. 1 compares the different methods of constrained optimization for Tabu Search, GA, and GALS for the MO-EC problem using the small simulation size. Heuristics are terminated after six hours of heuristic execution time. The heuristics show similar trends for the methods of constrained optimization. All heuristics and methods of constrained optimization produced solutions that met the energy-robustness constraint. The dynamic and static penalty functions can sometimes prefer infeasible solutions with a large objective value over feasible solutions with a smaller objective value, based on the relative value of Ψ and the weighted distance from feasibility. This can help the heuristics obtain a better objective value by allowing exploration into the infeasible region and guiding the search towards a better feasible solution in the end. The "superiority of feasible solutions" and "limiting the search space" techniques always prefer feasible solutions over infeasible ones, hindering the ability of the heuristics to accept high makespan-robustness solutions that may barely violate the energy-robustness constraint and eventually guide the heuristics to better feasible solutions, which led to worse results than the penalty function techniques.

The static penalty function must have a high enough penalty weight so that a feasible solution is found, but small enough such that the heuristics sometimes allow infeasible solutions to be accepted. We experimented with setting c equal to 0.5, 1, 2, and 5, and found the best results when setting c equal to 2, which allowed the heuristics to accept some infeasible solutions but to mostly prefer feasible ones.

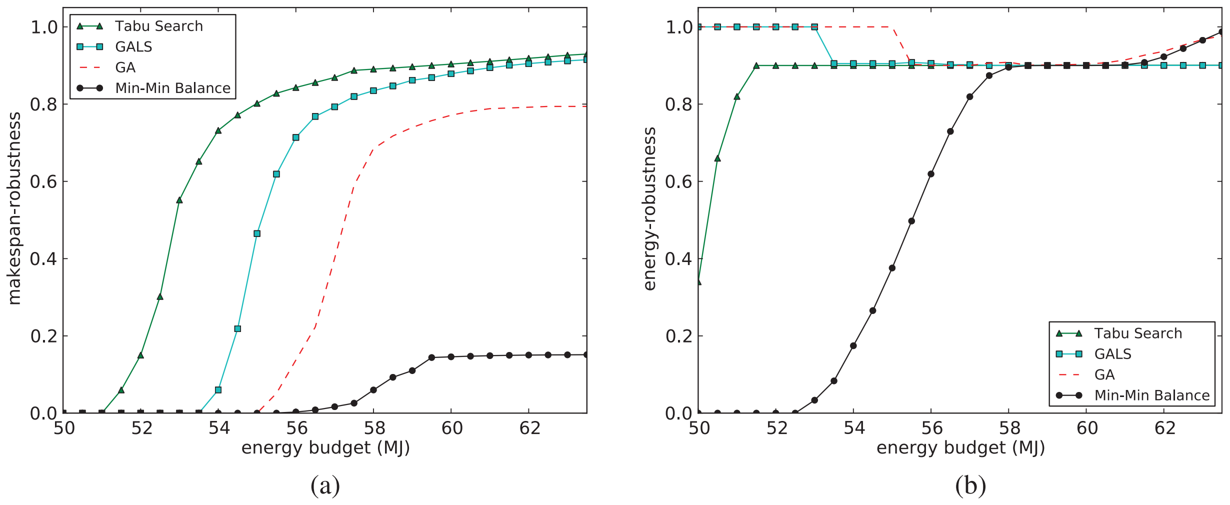
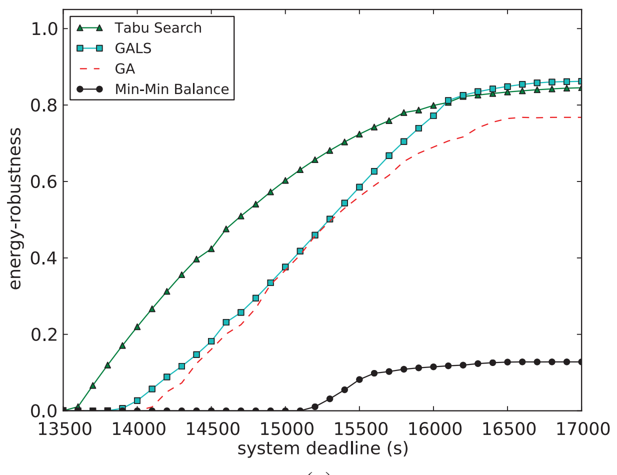
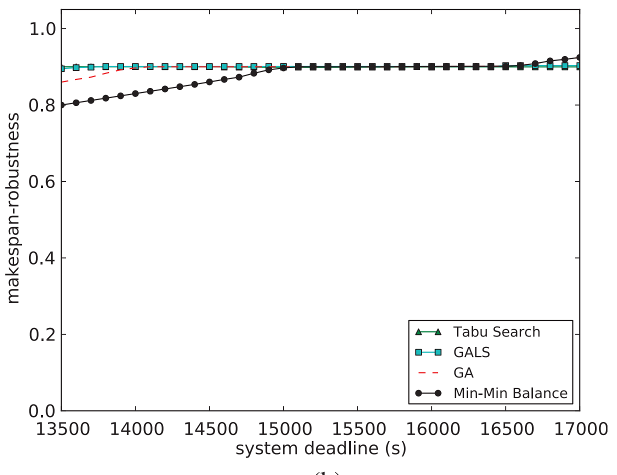


Fig. 2. Varying the energy budget for MO-EC: (a) makespan-robustness, and (b) energy-robustness (25 node system, 458 total cores, and 10,000 tasks). The system deadline was set to 15,500 seconds, the energy-robustness constraint was set to 90 percent. The heuristics were terminated after 6 hours of execution time.



(a)



(b)

Fig. 3. Varying the system deadline for EO-MC: (a) energy-robustness, and (b) makespan-robustness (25 node system, 458 total cores, and 10,000 tasks). The energy budget was set to 58 MJ and the makespan-robustness constraint was set to 90 percent. The heuristics were terminated after six hours of execution time.

Compared to the static penalty function, the dynamic penalty function is able to fine tune the penalty weight over the course of the search by adapting the penalty weight after each iteration of Tabu Search or generation of GALS and GA, resulting in better solutions than the static penalty function.

The "limiting the search space" technique is the same as the "superiority of feasible solutions" for Tabu Search, as Tabu Search starts with a Min-Min Balance solution that is feasible for the deadline and energy budget considered. However, for GALS and GA, the "limiting the search space" technique performs the worst as the initial population is generated with only feasible solutions, causing the chromosomes to lack diversity and hindering the benefits of the crossover operator. The results for EO-MC show the same trends as Fig. 1 across constrained optimization techniques. Because the dynamic penalty function performed best, we use this as our method of constrained optimization for other experiments.

Figs. 2 and 3 show the results of the heuristics at handling different values for the energy budget (Δ) and deadline (δ) for the small simulation size. Fig. 2 shows the results of the heuristics when maximizing makespan-robustness (Fig. 2a) with an energy-robustness constraint (Fig. 2b) when varying the energy budget (i.e., varying the difficulty of meeting the energy constraint). Fig. 3 shows the results of the heuristics when maximizing energy-robustness (Fig. 3a) with a makespan-robustness constraint (Fig. 3b) when varying the deadline. The energy-robustness and makespan-robustness constraints (η and Γ) were set to 90 percent.

For the MO-EC problem, Fig. 2 shows how well the heuristics perform at exploiting the trade-off between energy-robustness and makespan-robustness to sacrifice the probability of meeting the deadline (makespan-robustness) to meet the energy constraint. For this experiment, each of the 96 trials were executed with the deadline set to 15,500 seconds and the energy budget set to a value between 50 MJ and 64 MJ in 0.5 MJ increments. Each trial of the Tabu Search, GALS, and GA heuristics was terminated after six hours. Within this time, Fig. 2a shows that Tabu Search is able to obtain the best solutions at all energy budgets. Tabu Search and GALS are able to outperform GA in the allotted time as the intelligent local search operators used in Tabu Search and GALS are able to quickly identify moves that improve the solution, whereas the GA must rely on random genetic search which can go through several generations before finding better solutions. Tabu Search focuses on performing many short-hops on one solution and escaping local optima through long-hops, which outperforms the GALS that performs fewer short-hops (i.e., iterations of local search) on many different solutions over the course of the search and relies on random crossover and mutation to escape local optima.

We can observe that Tabu Search is able to achieve non-zero makespan-robustness and meet the energy-robustness contraint at energy budgets as small as 51 MJ, due to the local search operators intelligently assigning tasks to execute in low-power P-states on high-ranked nodes. In Fig. 2b, at energy budgets up to 53 MJ for GALS and 55 MJ for GA, GALS and GA return solutions of 1.0 energy-robustness (feasible) but 0.0 makespan-robustness. At these tight energy budgets, the Min-Min and Min-Min Balance seeds and randomly generated solutions of GALS and GA have very poor energy-robustness and are therefore highly

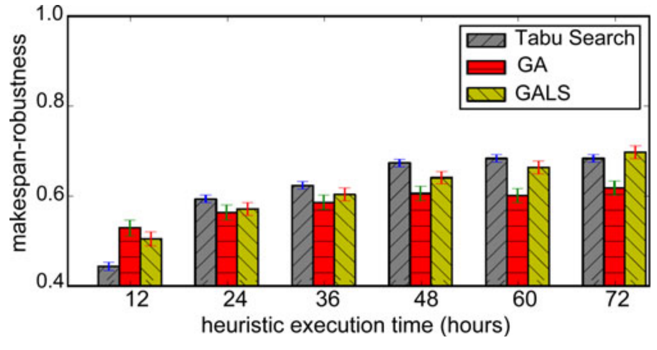
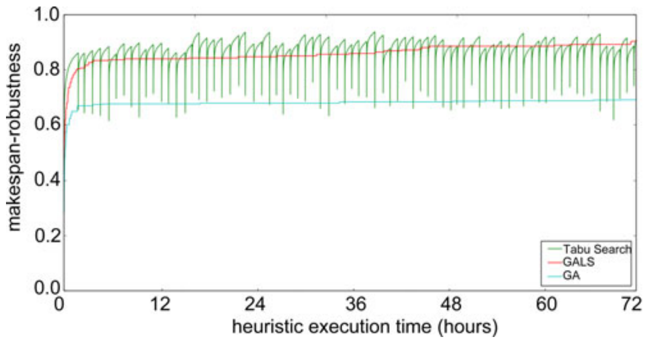
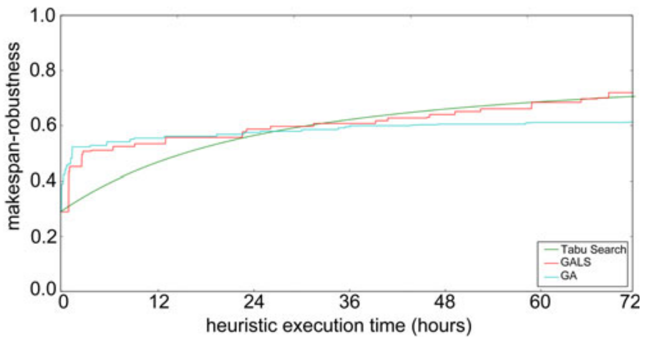


Fig. 4. Comparison of Tabu Search, GA, and GALS heuristics over 72 hours of execution time for MO-EC using the large simulation size (250 nodes, 4,580 total cores, and 100,000 tasks). The system deadline was set to 13,500 seconds, the energy budget was set to 580 MJ, and energy-robustness constraint was set to 90 percent.



(a)



(b)

Fig. 5. Progress of Tabu Search, GALS, and GA for MO-EC over 72 hours of heuristic execution time for the (a) small simulation size (25 nodes, 458 total cores, 10,000 tasks), and (b) large simulation size (250 nodes, 4,580 total cores, and 100,000 tasks). For the small simulation size, the system deadline was set to 15,500 seconds, the energy budget was set to 58 MJ, and the energy-robustness constraint was set to 90 percent. For the large simulation size, the system deadline was set to 13,500 seconds, the energy budget was set to 580 MJ, and energy-robustness constraint was set to 90 percent.

penalized, so the population quickly converges to solutions similar to the Min-Energy solution that has 1.0 energy-robustness and poor makespan-robustness. Also, when the energy budget is set to greater than approximately 60 MJ, the energy-robustness exceeds our set constraint of 0.9 for the GA and Min-Min Balance heuristics, indicating that when given a large energy budget, GA and Min-Min Balance are able to achieve feasible solutions but unable to use all available energy to increase makespan-robustness as desired. This is because the GA and Min-Min Balance heuristics do not incorporate the *P-state reassignment* operator that greedily assigns tasks to run in faster P-states until all available energy is used (e.g., until energy-robustness equals the constraint of 0.9). At higher energy budget values (over 60 MJ), GALS and Tabu Search have similar performance because the energy budget becomes easy enough that the energy-robustness constraint can be achieved when most tasks are running in the highest-power P-state (P0) (through the *P-state reassignment* operator for MO-EC), thus both heuristics can attain high makespan-robustness values while meeting the energy-robustness constraint.

Fig. 3 compares results of heuristics for EO-MC when varying the deadline (i.e., making the makespan-robustness constraint more or less difficult to attain). For this experiment, the energy budget is fixed at 58 MJ and system deadline is varied between 13,500 and 17,000 seconds. Again, we can observe that Tabu Search and GALS outperform the GA, indicating the significance of the local search operators for the smaller simulation size. Tabu Search outperforms GALS at most deadline values, as the benefit associated with improving one solution through using many short-hops and escaping local optima with long-hops exceeds the benefit associated with performing fewer local search iterations on numerous solutions and relying on random crossover and mutation to escape local optima, as in GALS. At higher deadline values (over 16,200 s), GALS and Tabu Search have similar performance because the deadline becomes easy enough that the makespan-robustness constraint can be achieved when most tasks are running in the lowest-power P-states (through the *P-state reassignment* operator for EO-MC), thus both heuristics can attain high energy-robustness values while meeting the makespan-robustness constraint.

The 25 node (458 total cores) platform is relatively small for a modern HPC system, so we also experimented with our heuristics on a larger simulated platform consisting of 250 nodes (4,580 total cores) and 100,000 tasks. Fig. 4 compares results of Tabu Search and GALS over 72 hours of heuristic execution time for MO-EC. With the smaller simulation size of only 25 nodes and 10,000 tasks, Tabu Search often achieved the best results (see Figs. 1, 2, 3). However, Fig. 4 shows that for 12 hours of heuristic execution time for the larger simulation size, GA outperforms both Tabu Search and GALS, and GALS outperforms Tabu Search. The local search operators used by Tabu Search and GALS are not as effective on the large simulation size, and there are two primary reasons: (1) the local search operators used by Tabu Search and GALS take considerably longer to execute when having to examine considerably more cores and tasks to intelligently swap and reassign tasks and P-states, and (2) local search operators only change one task and/or P-state assignment each, which can result in smallimprovements per iteration when considering 100,000 tasks instead of only 10,000 tasks. Genetic search (crossover and mutation) can change numerous task and P-state assignments per generation, which can lead to large improvements early in the heuristic. Over time, however, the randomness of the genetic search becomes less effective and improvements become incremental, leading the intelligent choices made by the Tabu Search and GALS to outperform GA. To help illustrate

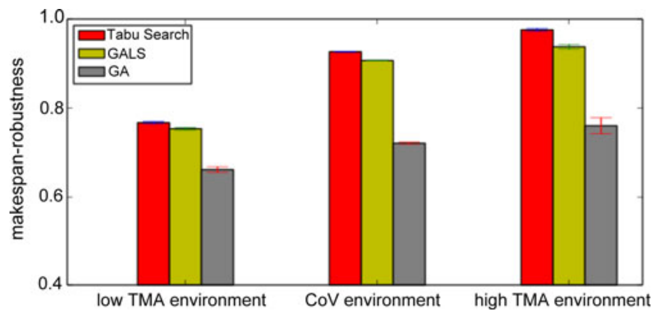


Fig. 6. Comparison of Tabu Search, GALS, and GA across heterogeneous environments with varying TMA for MO-EC using the small simulation size (25 nodes, 458 total cores, and 10,000 tasks). The system deadline was set to 15,500 seconds, the energy budget was set to 58 MJ, and the energy-robustness constraint was set to 90 percent.

these observations, Fig. 5 compares the progress of one trial of Tabu Search, GALS, and GA over 72 hours of heuristic execution time using the small simulation size (Fig. 5a) and large simulation size (Fig. 5b). On the small simulation size (Fig. 5a), Tabu Search is able to perform approximately 350,000 total iterations of local search over the 72 hours of heuristic execution time, however on the large simulation size (Fig. 5b) Tabu Search is only able to perform about 30,000 total iterations of local search, due to the increased time it takes for the short-hop operators to identify intelligent assignments. Fig. 5a shows Tabu Search being very effective on the small simulation size compared to GALS and GA, as the intelligent short-hop operators are fast and can focus on improving one solution until a long-hop is performed (the sharp decreases in makespan-robustness). The genetic search of GALS and GA leads to large improvements early, but the genetic search becomes less effective as the population becomes less diverse over time. GALS is able to perform few local search iterations on many different solutions, leading to a gradual improvement over time. Fig. 5b shows that with the large simulation size, Tabu Search does not perform any long-hops over 72 hours of heuristic execution time, meaning that the local search has yet to meet the termination criteria. We can see that the genetic search of GALS and GA is able to perform large jumps to better solutions early compared to the gradual improvement of Tabu Search due to crossover being able to change numerous assignments per generation through random recombination of chromosomes. The genetic search becomes less effective as the population converges, and Tabu Search obtains similar results to that of GALS after approximately 24 hours.

Fig. 6 compares the makespan-robustness results for MO-EC using our Tabu Search, GALS, and GA heuristics when the computing environment has high and low heterogeneity to evaluate the effectiveness of our heuristics across different heterogeneous environments. One measure for characterizing the heterogeneity of a computing environment is *t*ask *m*achine *a*ffinity (TMA) [1]. TMA captures the degree to which some tasks are better suited to run on specific machines. In an environment with low TMA, typically a node that is faster for one task is typically faster for all tasks. In contrast, an environment with high TMA contains nodes that are better-suited for some tasks, but other machines are better-suited for different tasks. We use methods detailed in [19] to modify the mean task execution time values from the CoV method (Appendix A, available in the online supplemental material) to create high and low TMA environments.

Fig. 6 shows all heuristics performing better on environments with high TMA than low TMA. In the environment with low TMA, all tasks execute the fastest on cores in the fastest node, leaving many tasks executing on subpar nodes when the workload is load balanced. In the high TMA environment, different tasks execute fastest on different nodes, leaving many tasks assigned to their fastest nodes when the workload is load balanced. We can also observe that in the high TMA environment, the makespan-robustness of Tabu Search and GALS exceed the performance of the GA by far more than in the low TMA environment. This is because the *task swap* and *task reassignment* local search operators employed by Tabu Search and GALS move tasks onto nodes that are better according to the mean rank matrix. In a high TMA environment, this makes the *task swap* and *task reassignment* operators very effective because placing tasks on their best-ranked nodes often leads to natural load balancing, giving high makespan-robustness. In the heuristic execution time given, Tabu Search and GALS greatly outperform the random combinations produced by GA. In the low TMA environment, placing tasks on better-ranked nodes does not load balance, leading to Tabu Search and GALS giving performance closer to that of the GA.

In summary, we found that the dynamic penalty function served as the most-effective constrained optimization technique (Fig. 1). Also, our Tabu Search heuristic gave the best results among the heuristics for the small simulation size (Figs. 2 and 3) and the large simulation size when given at least 24 hours to execute (Figs. 4 and 5). Tabu Search was also able to provide the best results in environments with low and high heterogeneity (Fig. 6).

8 Conclusions

In this paper, we design energy-aware resource allocation techniques to address two challenges that appear in today's data centers: (a) trying to optimize the makespan when subject to an energy budget constraint, and (b) trying to optimize energy consumption when subject to a makespan deadline. This problem becomes more complex when the execution times are modeled stochastically rather than deterministically. We develop probabilistic measures for both makespan and energy consumption, which we call makespan-robustness and energy-robustness. Makespan-robustness is the probability of meeting a makespan deadline, and energy-robustness is the probability of meeting an energy budget.

We approach this problem through the design of energy-aware resource allocation techniques incorporated with methods of constrained optimization from the literature. We compared the methods of constrained optimization using our Tabu Search, GALS, and GA heuristics. For a small simulation size, the intelligent search techniques of Tabu Search in combination with the dynamic penalty function outperformed the other resource allocation methods within the computation time given to execute the heuristics. For the large simulation size, however, the computation time overhead of our intelligent local search operators caused GA and GALS to outperform Tabu Search unless Tabu Search is given at least 24 hours to execute. The comparison of heuristics and constrained optimization techniques revealed great potential for our Tabu Search and GALS heuristics when combined with the dynamic penalty function for managing compute resources in an energy-aware manner for both deadline-constrained and energy-constrained systems.

For future work, we would like to consider workloads that consist of mostly compute-intensive or mostly memory-intensive tasks, which may require different techniques in addition to DVFS (such as consolidation) to reduce energy consumption. In addition, as multicore processors increase in number of cores, the effects of shared caches can have a pronounced impact on the execution time (and energy consumption) of tasks. Designing models and resource allocation techniques that account for these effects would be a necessity.

We also would like to consider tasks that have dependencies (which would require modeling precedence constraints and the numerous components of communication times). Designing new heuristics to adhere to the precedence constraints would be required. It is likely that the execution times of the new heuristics would be longer than for the independent task case. Because of the uncertainties in both execution time and communications, the complexity of the problem will increase as well as the design of simulations used to evaluate the new techniques.

ACKNOWLEDGMENTS

The authors thank T. Hansen and M. Amini Salehi for their valuable comments on this work. This research was supported by the National Science Foundation (NSF) under grant numbers CNS-0615170, CNS-0905399, 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. A preliminary version of portions of this material appear in the 2011 ACS/IEEE International Conference on Computer Systems and Applications [3] and the 2013 Workshop on Power and Energy Aspects of Computation [27].

REFERENCES

[1] A. M. Al-Qawasmeh, A. A. Maciejewski, R. G. Roberts, and H. J. Siegel, "Characterizing task-machine affinity in heterogeneous computing environments," in *Proc. 20th Heterogeneity Comput. Workshop*, May 2011, pp. 33–43.

[2] S. Ali, A. A. Maciejewski, and H. J. Siegel, "Perspectives on robust resource allocation for heterogeneous parallel systems," in *Handbook of Parallel Computing: Models, Algorithms, and Applications*, S. Rajasekaran, and J. Reif, Eds., Boca Raton, FL, USA: Chapman & Hall/CRC Press, pp. 41.1-41.30.

[3] J. Apodaca, D. Young, L. Briceño, J. Smith, S. Pasricha, A. A. Maciejewski, H. J. Siegel, S. Bahirat, B. Khemka, A. Ramirez, and Y. Zou, "Stochastically robust static resource allocation for energy minimization with a makespan constraint in a heterogeneous computing environment," in *Proc. 9th ACS/IEEE Int. Conf. Comput. Syst. Appl.*, Dec. 2011, pp. 22–31.

[4] T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, R. F. Freund, D. Hensgen, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, Jun. 2001.

[5] L. Briceño, H. J. Siegel, A. A. Maciejewski, M. Oltikar, J. Brateman, J. White, J. Martin, and K. Knapp, "Heuristics for robust resource allocation of satellite weather data processing onto a heterogeneous parallel system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 11, pp. 1780–1787, Nov. 2011.

[6] L. Briceño, J. Smith, H. J. Siegel, A. A. Maciejewski, P. Maxwell, R. Wakefield, A. Al-Qawasmeh, R. C. Chiang, and J. Li, "Robust static resource allocation of DAGs in a heterogeneous multicore system," *J. Parallel Distrib. Comput.*, vol. 73, no. 12, pp. 1705–1717, Dec. 2013.

[7] H. J. Choi, D. O. Son, S. G. Kang, J. M. Kim, H.-H. Lee, and C. H. Kim, "An efficient scheduling scheme using estimated execution time for heterogeneous computing systems," *J. Supercomput.*, vol. 65, no. 2, pp. 886–902, Aug. 2013.

[8] F. M. Ciorba, T. Hansen, S. Srivastava, I. Banicescu, A. A. Maciejewski, and H. J. Siegel, "A combined dual-stage framework for robust scheduling of scientific applications in heterogeneous environments with uncertain availability," in *Proc. 21st Heterogeneity Comput. Workshop*, May 2012, pp. 187–200.

[9] CSU Information Science and Technology Center. (2013). iSTeC Cray High Performance Computing (HPC) System [Online]. Available: http://istec.colostate.edu/istec_cray

[10] A. Doğan and F. Ozgüner, "Genetic algorithm based scheduling of meta-tasks with stochastic execution times in heterogeneous computing systems," *Cluster Comput.*, vol. 7, no. 2, pp. 177–190, Apr. 2004.

[11] F. Glover, "Tabu search, part I," *ORSA J. Comput.*, vol. 1, pp. 190–206, 1989.

[12] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: Research problems in data center networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 68–73, Jan. 2009.

[13] J. L. Hennessy, and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann, 2011.

[14] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation Std. (2011). *Adv. Configuration and Power Interface Specification*, rev. 5.0 [Online]. Available: http://www.acpi.info

[15] J. L. Hodges and E. L. Lehmann, *Basic Concepts of Probability and Statistics*. Philadelphia, PA, USA: SIAM, 2005.

[16] O. H. Ibarra, and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *J. Assoc. Comput. Mach.*, vol. 24, no. 2, pp. 280–289, Apr. 1977.

[17] M. A. Iverson, F. Ozgüner, and L. Potter, "Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment," *IEEE Trans. Comput.*, vol. 48, no. 12, pp. 1374–1379, Dec. 1999.

[18] Y.-S. Kee, K. Yocum, A. A. Chien, and H. Casanova, "Robust resource allocation for large-scale distributed shared resource environments," in *Proc. 15th Int. Symp. High Performance Distributed Comput.*, Jun. 2006, pp. 341–342.

[19] B. Khemka, R. Friese, S. Pasricha, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, S. Powers, M. M. Hilton, R. Rambharos, and S. Poole, "Utility maximizing dynamic resource management in an oversubscribed energy-constrained heterogeneous computing system," *Elsevier J. Sustainable Comput.: Informat. Syst.*, accepted to appear.

[20] A. Khokhar, V. Prasanna, M. Shaaban, and C.-L. Wang, "Heterogeneous computing: Challenges and opportunities," *IEEE Comput.*, vol. 26, no. 6, pp. 18–27, Jun. 1993.

[21] S.-G. Kim, C. Choi, H. Eom, H. Y. Yeom, and H. Byun, "Energy-centric DVFS controlling method for multi-core platforms," in *Proc. SC Companion: High Performance Comput., Netwo. Storage Anal.*, Jun. 2012, pp. 685–690.

[22] J. Koomey. (2011). Growth in data center electricity use 2005 to 2010, Analytics Press, Tech. Rep. [Online]. Available: http://www.analyticspress.com/datacenters.html

[23] D. Li and J. Wu, "Energy-aware scheduling for frame-based tasks on heterogeneous multiprocessor platforms," in *Proc. 41st Int. Conf. Parallel Process.*, Sep. 2012, pp. 430–439.

[24] Y. A. Li, J. K. Antonio, H. J. Siegel, M. Tan, and D. W. Watson, "Determining the execution time distribution for a data parallel program in a heterogeneous computing environment," *J. Parallel Distrib. Comput.*, vol. 44, no. 1, pp. 33–52, Jul. 1997.

[25] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," *J. Parallel Distrib. Comput.*, vol. 59, no. 2, pp. 107–131, Nov. 1999.

[26] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: Eliminating server idle power," in *Proc. Int. Conf. Archit. Support Program. Language Operating Syst.*, Mar. 2009, pp. 205–216.

[27] M. Oxley, S. Pasricha, H. J. Siegel, and A. A. Maciejewski, "Energy and deadline constrained robust stochastic static resource allocation," in *Proc. Workshop Power Energy Aspects Comput.*, Sep. 2013, p. 10.

[28] J. F. Pineau, Y. Robert, and F. Vivien, "Energy-aware scheduling of bag-of-tasks applications on master-worker platforms," *Concurrency Comput.: Practice Experience*, vol. 23, no. 2, pp. 145–157, Feb. 2011.

[29] F. Pinel, J. Pecero, S. Khan, and P. Bouvry, "Energy-efficient scheduling on milliclusters with performance constraints," in *Proc. IEEE/ACM Int. Conf. Green Comput. Commun.*, Aug. 2011, pp. 44–49.

[30] F. Pop, C. Dobre, and V. Cristea, "Genetic algorithm for DAG scheduling in grid environments," in *Proc. 5th IEEE Int. Conf. Intell. Comput. Commun. Process.*, Aug. 2009, pp. 299–305.

[31] D. Powell and M. M. Skolnick, "Using genetic algorithms in engineering design optimization with non-linear constraints," in *Proc. 5th Int. Conf. Genetic Algorithms*, 1993, pp. 424–431.

[32] G. Ritchie and J. Levine, "A hybrid ant algorithm for scheduling independent jobs in heterogeneous computing environments," in *Proc. 3rd Workshop UK Planning Scheduling Special Interest Group*, Dec. 2004.

[33] H.-P. Schwefel, *Evolution and Optimum Seeking*. New York, NY, USA: Wiley Interscience, 1995.

[34] V. Shestak, J. Smith, A. A. Maciejewski, and H. J. Siegel, "Stochastic robustness metric and its use for static resource allocations," *J. Parallel Distrib. Comput.*, vol. 68, no. 8, pp. 1157–1173, Aug. 2008.

[35] A. E. Smith and D. W. Coit, "Constraint-handling techniques—penalty functions," in *Handbook of Evolutionary Computation*, T. Back, D. Fogel, and Z. Michalewicz, Eds., Bristol, U.K.: Inst. Phy. Publishing and Oxford Univ. Press, 1997, ch. C5.2.

[36] Standard Performance Evaluation Corporation (SPEC). (2008). SPECpower_ssj2008 [Online]. Available: http://www.spec.org/power_ssj2008

[37] W. Sun, and T. Sugawara, "Heuristics and evaluations of energy-aware task mapping on heterogeneous multiprocessor platforms," in *Proc. Adv. Parallel Distrib. Comput. Models*, May 2011, pp. 599–607.

[38] D. Whitley, "The GENITOR algorithm and selective pressure: Why rank-based allocation of reproductive trials is best," in *Proc. 3rd Int. Conf. Genetic Algorithms*, Jun. 1989, pp. 116–121.

[39] Y. Wong, R. Goh, S. Kuo, and M. Low, "A Tabu Search for the heterogeneous DAG scheduling problem," in *Proc. 15th Int. Conf. Parallel Distrib. Syst.*, Dec. 2009, pp. 663–670.

[40] D. Xu, K. Nahrstedt, and D. Wichadakul, "QoS and contention-aware multi-resource reservation," *Cluster Comput.*, vol. 4, no. 2, pp. 95–107, Apr. 2001.

[41] O. Yeniay, "Penalty function methods for constrained optimization with genetic algorithms," *Math. Comput. Appl.*, vol. 10, no. 1, pp. 45–56, Jan. 2005.

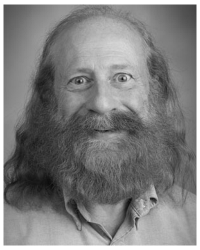
[42] B. D. Young, J. Apodaca, L. D. Briceño, J. Smith, S. Pasricha, A. A. Maciejewski, H. J. Siegel, B. Khemka, S. Bahirat, A. Ramirez, and Y. Zou, "Deadline and energy constrained dynamic resource allocation in a heterogeneous computing environment," *J. Supercomput.*, vol. 63, no. 2, pp. 326–347, Feb. 2013.

[43] W. Yuan, and K. Nahrstedt, "Energy-efficient soft real-time CPU scheduling for mobile multimedia systems," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 149–163, Dec. 2003.

Mark A. Oxley received the BS degree in computer engineering from the University of Wyoming. He is working toward the PhD degree and a research assistant in the Electrical and Computer Engineering Department at Colorado State University. His research interests include energy-aware, thermal-aware, and robust resource management techniques.

Sudeep Pasricha received the BE degree in electronics and communications from Delhi Institute of Technology in 2000, and the MS and PhD degrees in computer science from the University of California, Irvine, in 2005 and 2008, respectively. He is currently an associate professor in the Department of Electrical and Computer Engineering and also the Department of Computer Science at Colorado State University. He is a senior member of the IEEE and ACM.

Anthony A. Maciejewski received the BSEE, MS, and PhD degrees from the Ohio State University in 1982, 1984, and 1987, respectively. From 1988 to 2001, he was a professor of electrical and computer engineering at Purdue University, West Lafayette. He is currently a professor and the department head of Electrical and Computer Engineering at Colorado State University. He is a fellow of the IEEE. A complete vita is available at: http://www.engr.colostate.edu/ ~aam.



Howard Jay Siegel received the BS degree from MIT, and the MSE and PhD degrees from Princeton University. He was appointed the Abell Endowed chair distinguished professor of Electrical and Computer Engineering at Colorado State University in 2001, where he is also a professor of computer science. From 1976 to 2001, he was a professor at Purdue University. He is a fellow of the IEEE and ACM.

Jonathan Apodaca is an undergraduate studying computer science at Colorado State University. His research interests include resource management in heterogeneous computing systems.

Dalton Young received the MS degree in electrical engineering from the University of Kentucky in 2009. He is currently working toward the PhD degree in the Electrical and Computer Engineering Department at Colorado State University.

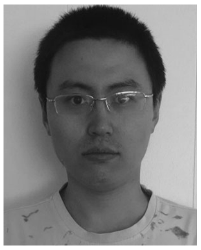
Luis D. Briceño received the BS degree in electrical and electronic engineering from the University of Costa Rica, and the PhD degree in electrical and computer engineering at Colorado State University. He is currently a component design engineer at Intel. His research interests include heterogeneous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

Jay Smith received the PhD degree in electrical and computer engineering from Colorado State University in 2008. He co-founded Lagrange Systems in 2012, and currently serves as the chief technical officer of the company. He has coauthored more than 30 peer reviewed articles in the area of parallel and distributed computing systems. In addition to his academic publications, while at I.B.M., he received more than 20 patents and numerous corporate awards for the quality of those patents. He left I.B.M. as a master inventor in 2008 to focus on high-performance computing at DigitalGlobe. There, he pioneered the application of GPGPU processing within DigitalGlobe. In addition to his position at Lagrange Systems, he serves as a research faculty member in the Electrical and Computer Engineering Department at Colorado State University. His research interests include high-performance computing and resource management. He is a member of both the IEEE and the ACM.

Shirish Bahirat received the the MS degree in electrical engineering from the University of Colorado, and the PhD degree in electrical and computer engineering and MBA from Colorado State University. He also possesses extensive experience in leading product design and development efforts for global high-tech companies creating leading-edge technology solutions. His research interests include multicore SOC and solid state memory technology.

Bhavesh Khemka received the BE degree in electrical and electronics engineering from Hindustan College of Engineering affiliated with Anna University, India, in 2009, and the PhD degree in electrical and computer engineering (ECE) from Colorado State University (CSU) in 2014. He is a postdoctoral research scholar in the Electrical and Computer Engineering (ECE) Department at Colorado State University (CSU). His research interests include fault-tolerant, energy-aware, and robust resource management in heterogeneous and distributed computing environments.

Adrian Ramirez received the BS degree in electrical engineering at Texas A&M University-Kingsville in 2007, and the MS degree in electrical engineering with a focus in robotics at Colorado State University in 2010. He is currently an Electrical Development Engineer in R&D at Covidien (scheduled to be acquired by Medtronic in 2015) where he specializes in battery management and switch-mode converters.



Yong Zou received the MS degree in software engineering from the University of Science and Technology of China. He is working toward the PhD degree in the Electrical and Computer Engineering Department at Colorado State University. His current research interests are in the areas of fault tolerance in network-on-chip, computer architecture, and embedded system design.