# Stochastically Robust Static Resource Allocation for Energy Minimization with a Makespan Constraint in a Heterogeneous Computing Environment

Jonathan Apodaca[#1], Dalton Young[*2], Luis Briceño[*3], Jay Smith[*4,$12],
Sudeep Pasricha[*5], Anthony A. Maciejewski[*6],Howard Jay Siegel[*7],
Shirish Bahirat[*8], Bhavesh Khemka[*9], Adrian Ramirez[*10], Young Zou[*11]

[#]*Department of Computer Science,* [*]*Department of Electrical & Computer Engineering, Colorado State University*
*Fort Collins, CO 80523-1373, USA*
{[1]jonathan.apodaca, [2]dalton.young,
[3]ldbricen, [4]james.t.smith_ii, [5]sudeep,
[6]aam, [7]hj, [8]shirish.bahirat, [9]bhavesh.khemka,
[10]adrian.ramirez, [11]yong.zou} @colostate.edu

[$]*DigitalGlobe*
*Longmont, CO 80503, USA*
[12]jtsmith@digitalglobe.com

***Abstract*—In a heterogeneous environment, uncertainty in system parameters may cause performance features to degrade considerably. It then becomes necessary to design a system that is robust. Robustness can be defined as the degree to which a system can function in the presence of inputs different from those assumed. In this research, we focus on the design of robust static resource allocation heuristics suitable for a heterogeneous compute cluster that minimize the energy required to complete a given workload. In this study, we mathematically model and simulate a heterogeneous computing system that is assumed part of a larger warehouse scale computing environment. Task execution times/energy consumption may vary significantly across different data sets in our heterogeneous cluster; therefore, the execution time of each task on each node is modeled as a random variable. A resource allocation is considered robust if the probability that all tasks complete by a system deadline is at least 90%. To minimize the energy consumption of a specific resource allocation, dynamic voltage frequency scaling (DVFS) is employed. However, other factors, such as system overhead (spent on fans, disks, memory, etc.) must also be mathematically modeled when considering minimization of energy consumption. In this research, we propose three different heuristics that employ DVFS to minimize energy consumed by a set of tasks in our heterogeneous computing system. Finally, a lower bound on energy consumption is provided to gauge the performance of our heuristics.**

## I. Introduction

Parallel and distributed systems may operate in an environment where certain system performance features degrade due to unpredictable circumstances, e.g., inaccuracies in estimated system parameters. Robustness can be defined as the degree to which a system can function correctly when the values of system parameters are different from those expected.

This research was supported by the National Science Foundation under grant numbers CNS-0615170 and CNS-0905339, and by the Colorado State University George T. Abell Endowment.

In this study, we mathematically model a heterogeneous computing system and run simulations to evaluate potential power-consumption performance improvement using dynamic voltage and frequency scaling (**DVFS**). Our compute cluster is assumed to be part of a larger warehouse scale computing environment [6] such that even a modest improvement in energy consumption within each single cluster will result in large power savings at the warehouse level. In this data center, each cluster consists of a single rack of heterogeneous (different types of) compute nodes, where the performance across nodes in the cluster varies substantially.

Performance is defined in terms of the time required to complete the provided workload. Furthermore, the performance of individual machines in this cluster is assumed to be inconsistent [4]. That is, if machine $A$ is faster than machine $B$ for a given task, that does not imply that machine $A$ is faster for all tasks.

By utilizing a heterogeneous mix of machines it may be possible to increase the power efficiency of the data center. Further, through careful allocation of resources, it may be possible to reduce the energy required to complete a given workload by its time constraint, thus reducing the overall operating costs of the data center. To minimize energy consumption of a specific resource allocation, DVFS is employed. Because the performance and power efficiency of compute nodes in this environment are heterogeneous, we require resource allocation heuristics that are capable of leveraging these performance differences to reduce energy consumption. The goal of resource allocation heuristics in this environment is to allocate a collection of independent tasks to machines so that we minimize the energy required to complete the workload, while ensuring that there is a high probability that the entire workload completes by a system provided deadline.

That is, the challenge is in attempting to minimize expected energy consumption while not ignoring a conflicting constraint on the execution time of the workload (i.e., there must be a high probability that the entire workload completes by a deadline).

The major contributions of this paper are: (a) a stochastic robustness model for this environment, (b) three heuristics that generate resource allocations that have a high probability of processing a given workload by the provided deadline while minimizing energy consumption, and (c) a lower bound on energy consumption to gauge how the heuristics perform.

In the next section, we define the system model of the compute cluster and workload. Based on this system model, in Section III, we derive a robustness measure suitable for this environment that quantifies the probability of meeting our makespan constraint for a given resource allocation. Three heuristics used to generate resource allocations that attempt to minimize energy consumption while satisfying a robustness constraint are proposed in Section IV. A lower bound on energy consumption is given in Section V. Sections VI and VII discuss the simulation setup and results collected for each heuristic. Section VIII provides a discussion of related work. Finally, conclusions are presented in Section IX.

## II. System Model

### A. Compute Nodes and Power

A cluster consists of $\boldsymbol{N}$ compute nodes. Each compute node $i$ consists of $\boldsymbol{n_i}$ multicore processors, where the value of $n_i$ varies from one to four. Each multicore processor $j$ in compute node $i$ has $\boldsymbol{n_{ij}}$ cores, where $n_{ij}$ also varies from one to four. Figure 1 shows the hierarchy of nodes, multicore processors, and cores. For convenience, we let $\boldsymbol{N_{total}}$ denote the total number of cores in the system ($N_{total} = \sum_{i=1}^{N} \sum_{j=1}^{n_i} n_{ij}$).

As defined in the Advanced Configuration and Power Interface (**ACPI**), in our system there are five available processor performance states (**P-states**) [2], and each core within a node has an interface for switching P-states. While in current systems all cores in a processor will have the same P-state, to make our problem more complex and interesting, we consider the future direction of allowing each core to be set to a P-state individually. Each P-state corresponds to a voltage level and clock frequency, and is subject to a corresponding power consumption for maintaining that voltage level. To minimize overhead of switching from one P-state to another, we assume that once a core starts executing a task in a particular P-state, it may not change P-states while the task is executing.

As power is increased to a core the performance of that core also increases, i.e., task execution times decrease. Let $\boldsymbol{P}$ denote the set of all available P-states where $P_0$ corresponds to the base P-state that has the lowest power consumption (and lowest performance for that multicore processor). The power consumed by state $P_\pi$ in compute node $i$ is denoted $\boldsymbol{\rho_i^{(\pi)}}$, where $\pi \in \{0, 1, 2, 3, 4\}$. When all multicore processors are idle within a compute node, the node transitions to a hibernate state, and that the node consumes no power. Because this is a

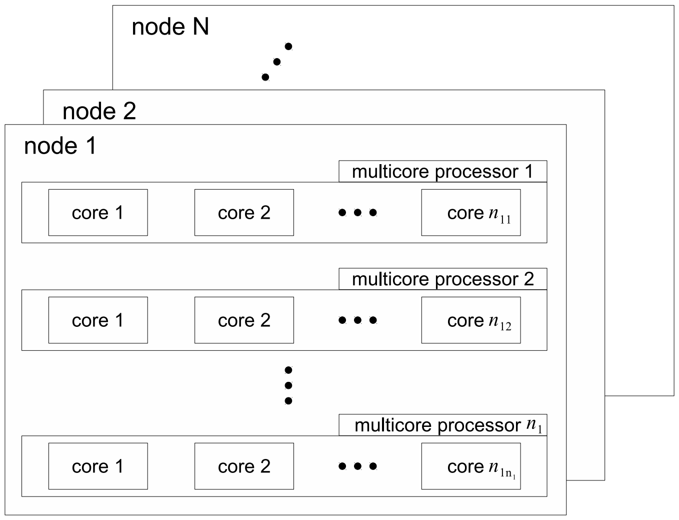

Fig. 1. Hierarchy of nodes, multicore processors, and cores.

static environment (see II-B), and all tasks are independent, a core is not idle until all tasks assigned to it have completed, after which the core shuts off. Similarly, a multicore processor is not idle until the last core within that multicore processor turns off, after which the multicore processor turns off.

In addition to the heterogeneity in performance, the compute nodes of this cluster are heterogeneous in the power efficiency of their respective power supplies. Power efficiency relates the total power provided by the power supply to the actual power that is supplied to the computing system. For example, for every 80 watts of power provided by a non-perfect power supply with an efficiency of 80%, that power supply would require 100 watts of power. In our cluster, the efficiency of power supplies varies from 75% efficient to at most 90% efficient. Let $\boldsymbol{\epsilon_i}$ denote the power efficiency of the power supply in compute node $i$.

### B. The Workload

The workload in this environment is a static collection of $\boldsymbol{T}$ independent tasks (i.e., all tasks to be executed by the system are known *a priori*), that must be completed before a provided a deadline, denoted $\boldsymbol{\delta}$. We consider a static production environment where the complete set of tasks to be executed is known in advance, and that task execution times may be data dependent (e.g., [21]). There is uncertainty in task execution times and this uncertainty can be modeled using a normal distribution. The mean and variance of this normal distribution are determined by historical, experimental, or analytical techniques [15], [21], [25]. The means of these distributions will be used to calculate the expected makespan of the workload in the next subsection and expected energy consumption in Subsection II-D for a given resource allocation. In this system, the expected task execution times and task variances vary across heterogeneous nodes. That is, we model the execution time of each task on each compute node in each P-state as a random variable and we are given a probability density function (**pdf**) describing the probability of each of the possible execution times.

As common in the research allocation literature, we consider a non-multitasking environment (e.g., [9], [12]). Also, potentially complex interactions between tasks executing on neighboring cores that share resources, such as cache memory interactions, are not explicitly modeled (but may be modeled with the normal distributions of the task execution times).

### C. Calculating Expected Makespan

Given a resource allocation calculating the expected makespan requires that we first determine the expected finishing time of each compute node given this resource allocation. Because the tasks are independent, once we have established the expected finishing time of each compute node, we can find the expected makespan as the maximum of all of the expected node finishing times.

Let the set $\boldsymbol{T_{ijk}}$ denote all tasks in $T$ that have been assigned to core $k$ on multicore processor $j$ of compute node $i$, and that $\boldsymbol{t_{ijk}} \in T_{ijk}$. Let the function **pstate**$(\boldsymbol{t_{ijk}})$ return the selected P-state for task $t_{ijk}$. Each entry in the expected time to compute (ETC) matrix, denoted **ETC**$\{\boldsymbol{t_{ijk}}, \textbf{pstate}(\boldsymbol{t_{ijk}})\}$, depends on the task to be executed, the core that is to execute it, and the selected P-state for the core. The expected finishing time of core $k$ on multicore processor $j$ in compute node $i$, denoted $\boldsymbol{F_{ijk}}$ is given as

$$F_{ijk} = \sum_{\forall t_{ijk} \in T_{ijk}} \text{ETC}\left\{t_{ijk}, \text{pstate}(t_{ijk})\right\}. \tag{1}$$

Therefore, the expected makespan of a resource allocation, denoted $\boldsymbol{\Delta}$, can be found as the maximum expected finishing time among all cores, multicore processors, and compute nodes, and is given as

$$\Delta = \max_{\forall k \in n_{ij}} \left\{ \max_{\forall j \in n_i} \left\{ \max_{\forall i \in N} F_{ijk} \right\} \right\}. \tag{2}$$

### D. Calculating Expected Energy Consumption

The total energy required to process the workload is determined by summing the power consumption from the start of the workload through the completion of the workload. Because the P-state of each core may differ throughout the course of the workload execution, we can find the power consumption for each core in each P-state and sum over the time spent in each P-state for that core. However, we would also like to account for the overhead power consumption of an active multicore processor (e.g., L3 cache) and an active compute node (e.g., disk drives, fans, RAM). For example, Figure 2 gives typical server component energy consumption for three sample systems and shows that system overhead can be a significant component of overall energy consumption [17]. System overhead can consume lower percentages with respect to total system energy, as seen in published reports such as SPECpower_ssj® 2008 [1].

Let $\boldsymbol{T^{\pi}_{ijk}}$ denote the subset of tasks assigned to core $k$ on multicore processor $j$ of compute node $i$ processed in P-state $\pi$, i.e., $T^{\pi}_{ijk} = \{\forall t_{ijk} \in T_{ijk} | \text{pstate}(t_{ijk}) = \pi\}$. Because this is a static resource allocation problem and all tasks are independent, all tasks on a given core can be reordered such that all tasks in P-state $P_0$ execute first, all tasks in $P_1$ execute next, and so on for all P-states. Thus, the expected energy spent by core $k$, within multicore processor $j$, in node $i$, in P-state $\pi$, denoted $\boldsymbol{S^{\pi}_{ijk}}$, is given as

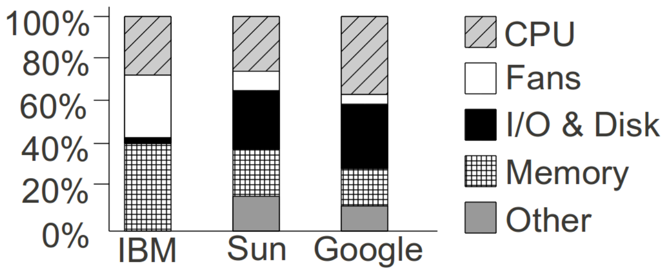

Fig. 2. Server component energy usage, based on [17].

$$S^{\pi}_{ijk} = \sum_{t_{ijk} \in T^{\pi}_{ijk}} \rho_i^{(\pi)} \text{ETC}\left\{t_{ijk}, \pi\right\}. \tag{3}$$

To determine the total expected energy consumption of a multicore processor, we wish to account for the overhead of shared resources on the multicore processor during the time that the multicore processor is active. Let $\boldsymbol{F_{ij}}$ be the maximum completion time among cores in multicore processor $j$ on node $i$, $\boldsymbol{F_i}$ be the maximum completion time among multiprocessors in node $i$, $\boldsymbol{\omega_i^{MP}}$ be the power overhead of shared resources for each multicore processor in compute node $i$, and $\boldsymbol{\omega_i^{node}}$ be the power overhead for compute node $i$. We can calculate the expected energy required to process the entire workload across all $N$ compute nodes, denoted $\zeta$, as

$$\begin{aligned}\zeta = \sum_{i=1}^{N} \frac{1}{\epsilon_i} \Bigg( & F_i \omega_i^{node} + \sum_{j=1}^{n_i} \left(F_{ij} \omega_i^{MP}\right) \\ & + \sum_{j=1}^{n_i} \sum_{k=1}^{n_{ij}} \sum_{\forall \pi \in P} S^{\pi}_{ijk} \Bigg)\end{aligned} \tag{4}$$

## III. Robustness

### A. Overview

Resource allocation decisions are often based on estimated values of task and system parameters, whose actual values are uncertain and may differ from available estimates [22]. A resource allocation can be considered "robust" if it can mitigate the impact of uncertainties in system parameters on a given performance objective [3]. That is, a robust resource allocation can guarantee a certain level of *system performance* under a wide range of conditions. Any claim of robustness for a given system must be able to answer the following three robustness questions [5]: (a) What behavior makes the system robust? (b) What are the uncertainties that the system is robust against? (c) How is system robustness quantified? These three questions help establish an intuitive meaning for the robustness of a system.

In this environment, a resource allocation can be considered robust if it completes the entire workload within the makespan constraint $\delta$. Recall that the expected execution time for each task on each machine is only an estimate and that the actual execution time may vary based on the data being processed. The stochastic robustness metric used in this paper is based on the concepts presented in [21], [23]. Thus, the robustness of a resource allocation can be quantified as the probability that the makespan constraint will be satisfied [21].

Our answers, then, to the three robustness questions presented above are: (a) a given resource allocation is robust if it completes the entire workload by the makespan constraint; (b) the system is robust against uncertainty in task execution times; and (c) the robustness of a resource allocation is quantified as the probability of completing a given workload by a system-provided deadline.

## B. Calculating Robustness

Because all tasks are independent and all cores can operate independently, the robustness of the system is the product of the probability for each core in the compute cluster to finish its assigned workload in at most $\delta$ time units. To simplify our simulation calculations, we assume, without loss of generality, that the execution time distributions for tasks are normally distributed, possibly having different means and variances. Convolution of two normally distributed random variables $\alpha$ and $\beta$ will produce a result that is also normally distributed, where the mean is given as the sum of the means of $\alpha$ and $\beta$ and the variance is given as the sum of the variances for $\alpha$ and $\beta$. Thus, we can use the expected finishing time computation of Equation 1 to find the mean of the finishing time distribution for each core. That is, given core $k$ on multicore processor $j$ in compute node $i$, the mean of the finishing time distribution is given as $F_{ijk}$. To find the variance, let $\boldsymbol{V[t_{ijk}, \pi]}$ denote the variance associated with task $t_{ijk}$ executed on compute node $i$, multicore processor $j$, core $k$, in P-state $\pi$. The variance, denoted $\sigma^2_{ijk}$, can be found as

$$\sigma^2_{ijk} = \sum_{\forall t_{ijk} \in T_{ijk}} V\left\{t_{ijk}, \text{pstate}(t_{ijk})\right\}. \tag{5}$$

Thus, the finishing time distribution can be expressed as $\mathcal{N}(F_{ijk}, \sigma^2_{ijk})$. The probability that this core will finish before $\delta$ can be found by converting the normal probability density function $\mathcal{N}(F_{ijk}, \sigma^2_{ijk})$ to a cumulative density function and finding the probability associated with a finishing time of $\delta$. Let $\textbf{CDF}(f, \delta)$ denote the probability of finishing before $\delta$ using the cumulative distribution function of $f$. The overall system robustness for a given resource allocation, denoted $\boldsymbol{\Psi}$, can be found by multiplying the robustness values for each individual core together, because tasks and cores are independent, and is found as

$$\Psi = \prod_{\forall i \in N} \prod_{\forall j \in n_i} \prod_{\forall k \in n_{ij}} \text{CDF}\left(\mathcal{N}(F_{ijk}, \sigma^2_{ijk}), \delta\right). \tag{6}$$

In this study we require resource allocations to meet a robustness constraint of $\psi \geq 90\%$.

## IV. Heuristics

### A. Overview

The goal of the heuristics is to produce resource allocations that minimize energy consumption while meeting a robustness constraint of at least 90% probability or greater of completing by the deadline $\delta$. Because this is a static resource allocation for a set of production applications known *a priori*, the resource allocation is derived off-line before any of the tasks begin execution. We adopted three different global resource allocation heuristics to this environment: a Tree Search heuristic, a Genetic Algorithm (GA) and, a Tabu search heuristic. We also present two two-phase greedy heuristics, Min-Min and Min-Max, that execute quickly and are used to seed some of the global search heuristics.

### B. Min-Min and Min-Max

The Min-Min and Min-Max heuristics (based on concepts in [7], [11], [16]) greedily assign tasks to cores, making one assignment per iteration, where all assignments use a single fixed P-state. Initially, all tasks are "unmapped" (placed in the **unmapped batch**). At each iteration of Min-Min or Min-Max, the heuristic uses two phases to choose a single task-to-core assignment. In the first phase of both heuristics, the minimum expected completion time (**MECT**) core is found for each task in the unmapped batch. In the second phase, Min-Min chooses the task-core combination from phase one that yields the smallest overall MECT for assignment and removes the corresponding task from the unmapped batch. Min-Max instead chooses the task-core combination from phase one that yields the largest overall MECT for assignment and removes the corresponding task from the unmapped batch. All core ready times are then updated, and the heuristic begins another iteration in phase one. Both heuristics iterate until all tasks have been mapped (i.e., the unmapped batch is empty).

### C. Tree Search

In our Tree Search (**TS**) heuristic based on [24], [27], the set of all possible solutions is conceptually organized as a tree, where each node in the tree corresponds to a task, machine, and P-state assignment. Let $\boldsymbol{T'}$ denote a list of all tasks in the order that the Min-Min heuristic would map them using the highest P-state ($P_4$), $\boldsymbol{|P|}$ denotes the number of available P-states, and recall that $N_{total}$ is the total number of cores in the system. The tree has $|T'| + 1$ levels, and each node has $m = |P| \times N_{total}$ children. The root node of the tree represents a null solution. There are $N_{total} \times |P|$ children for each node corresponding to all of the cores at each possible P-state, and a node at level $\boldsymbol{L}$ corresponds to mapping task $T'_L$ to the corresponding core and P-state.

The Tree Search starts at the root node, marking it as unexplored, and works its way down the tree by iteratively expanding unexplored nodes. Only nodes that are both on the current level $L$ and marked unexplored are considered *active* and are eligible for expansion. An active node is expanded by marking each of its child nodes as unexplored and marking it as inactive, but this expansion procedure quickly increases the number of unexplored nodes. The heuristic execution time is directly correlated to the number of unexplored nodes: the execution time and memory requirements of the heuristic become infeasible if too many nodes in the tree are marked as unexplored. Therefore, our implementation utilizes a trimming scheme to limit the number of unexplored nodes to $\boldsymbol{\beta}$ nodes. In our experiments, the best results for Tree Search were achieved by setting $\beta$ to 40.

To trim the list of unexplored nodes, a cost function $\boldsymbol{z(n)}$ is applied to each node $\boldsymbol{n}$ in the unexplored list. The expected energy consumption of a partial mapping through node $n$ is denoted $\boldsymbol{\zeta(n)}$. Let $\boldsymbol{\Delta_n}$ denote the expected makespan through the current node $n$, let $\boldsymbol{L_n}$ denote the level of node $n$, and let $\boldsymbol{\tau}$ be a threshold indicating a desired upper bound on $\Delta$ (in our simulations, $\delta = 210$ and $\tau$ was empirically set to 145). Using these quantities, we define $z(n)$ as

$$z(n) = \Delta_n - \frac{L_n}{|T|} \cdot \tau. \tag{7}$$

Intuitively, $z(n)$ indicates how far "ahead" (when $z(n) < 0$) or "behind" (when $z(n) > 0$) the makespan of node $n$ is with respect to $\tau$ (assuming each task uses an equal portion of $\tau$). To trim, we create a list of unexplored nodes partitioned into two groups: 1) nodes with $z(n) < 0$ and 2) nodes with $z(n) \geq 0$. The first partition is therefore composed of all partial mappings that are "ahead" of $\tau$, and these are ordered with increasing energy consumption $\zeta(n)$. The second partition, located after the first partition, is composed of all partial mappings that are either on or "behind" schedule, and these are ordered with increasing $z(n)$. Out of this partitioned list, we keep the first $\beta$ nodes for further exploration, and delete the rest.

Once the last level of the tree is encountered, the node $\boldsymbol{n_{best}}$ (the leaf node with the lowest energy consumption that meets the robustness constraint) is selected for the final resource allocation. The allocation is determined by tracing the path of $n_{best}$ back up the tree and deducing the task to core/P-state assignments.

## *D. Genetic Algorithm*

Our genetic algorithm (GA) is based on the concepts in [10], [19], and attempts to probabilistically generate a resource allocation by incrementally altering existing solutions and removing poor ones. A fixed-size group of complete resource allocations forms the population on which the GA operates. These are random allocations that do not violate the robustness constraint, and we add allocations from Min-Min using each P-state to the population as seeds. The GA operates by probabilistically forming new allocations from existing allocations, and then sorting the new and previous allocations and keeping the best. The GA executes for a fixed number of iterations, denoted as $\boldsymbol{max_{iter}}$ (empirically set to 250,000), and then returns the best allocation it encountered.

The population on which the GA operates consists of chromosomes, where each chromosome represents a complete mapping of every task. Each task assignment, also called a **gene**, consists of a four-tuple denoting the node, multicore processor (MP), and core assignment for the task, as well as the P-state that the task will be executed. The chromosome length (number of genes) is equal to the number of tasks in the system.

Each chromosome in the initial population is generated as follows, so that the robustness constraint is not violated. For each task $t$, find the node $i$ that minimizes the expected *execution time* of task $t$ in P-state $P_4$, and map task $t$ to a random core $k$ within node $i$ if the expected *completion time* of core $k$ does not exceed a completion time threshold, denoted as $\boldsymbol{ct_{thresh}}$ (empirically set to 160). If $ct_{thresh}$ is violated, map task $t$ to another random core in node $i$, and repeat until either the completion time is less than $ct_{thresh}$ or there are no cores left in $i$ to try. If there are no cores left to try in $i$, map task $t$ to the next node that minimizes the expected *execution time*, repeating the above steps. If the task cannot be assigned to any core in the system without its completion time violating $ct_{thresh}$, the task is mapped to the core that gives its minimum expected completion time (this never happened during our simulations). This process is repeated until all tasks are mapped within a chromosome, and the entire initial population is generated by repeatedly creating chromosomes in the above manner. In addition to the generated chromosomes, the GA is seeded with five Min-Min chromosomes, one from each of the five P-states.

The GA limits the population size, denoted $\boldsymbol{|C|}$ (empirically set to 100), after each operation by ranking the chromosomes according to fitness value and eliminating the lowest-ranked chromosomes such that the population size remains fixed at its original size, similar to the Genitor heuristic [26]. Fitness is determined in two steps. First, chromosomes are separated into two groups: those that meet the robustness constraint, followed by those that do not. Any chromosome that meets the robustness constraint is more-fit than any chromosome that does not. Within each group, the chromosomes are sorted by their expected energy values (the expected energy needed to compute the workload according to the mapping in the chromosome). Therefore, the more-fit of two chromosomes within the same group is the one that consumes the least energy.

The crossover operation generates new genetic material by probabilistically swapping task assignments between chromosomes. The number of crossover operations during a single iteration of the GA is equal to the size of the population. To begin each crossover operation, two parents are randomly selected, and with probability $\boldsymbol{p_c}$ (determined empirically to be 0.005), two offspring are generated. That is, a uniform random number between zero and one is generated and compared with $p_c$. If the number is $\leq p_c$, then the crossover operation continues; otherwise it is aborted. The operation uses a two-point crossover operation to swap assignment information between parents. Two crossover points $x$ and $y$ are generated such that $x < y$ and $y < |T|$. Then, all of the task-to-core/P-state assignments in the first chromosome, from task $x$ to task $y$, are interchanged with the assignments from task $x$ to task $y$ from the second chromosome. The chromosomes generated by crossover are added to the population after all crossover operations have completed in an iteration, and the least-fit chromosomes are eliminated to bring the population size back to $|C|$.

There are two mutation operators that generate new genetic material by probabilistically altering task assignments: 1) task-assignment mutation, and 2) P-state mutation. In task-assignment mutation, each chromosome has a probability $\boldsymbol{p_{tm}}$ of being mutated (determined empirically to be 0.25). If a chromosome is selected for mutation, each gene in that chromosome has a probability $\boldsymbol{p_{tmg}}$ of being mutated (empirically set to 0.001). If a gene is mutated, the task corresponding to that gene is assigned to a random core and random P-state. The P-state mutation is very similar to task-assignment mutation, except that each chromosome has a probability $\boldsymbol{p_{pm}}$ of being mutated (experimentally determined to be 0.025), and each gene within a chromosome has a probability $\boldsymbol{p_{pmg}}$ (determined empirically to be 0.0005) of being assigned a random P-state (only the P-state is changed). The chromosomes generated by mutation are added to the population after all mutation operations have completed in an iteration, and the least-fit chromosomes are eliminated to bring the population size back to $|C|$.

Once the GA completes $max_{iter}$ iterations, the chromosome with the highest fitness is chosen as the resource allocation.

## E. Tabu

The Tabu heuristic based on concepts in [7], [13], [18] combines global search ("long-hops") and local search ("short-hops") to gain the benefits of both. Each solution is represented as in the GA seen in the previous subsection. Each unmodified long-hop is stored in a tabu list, indicating that the neighborhood represented by that particular long-hop may not be searched again. In our implementation of Tabu, two solutions are in the same neighborhood when 50% or greater of task-to-node assignments are the same between them. When a neighborhood has been sufficiently searched by short hops, long hops are used to escape local minima.

Multiple short-hops are executed after each long-hop to find the "local minimum near the long-hop" solution, with a maximum of $\boldsymbol{max_{SH}}$ short-hops per long-hop. Short-hops are small, incremental changes around long-hops. The heuristic executes a maximum number of long-hops, denoted $\boldsymbol{max_{LH}}$, and then returns the best allocation discovered. The values for $max_{SH}$ and $max_{LH}$ were empirically set to 6,000 and 100, respectively.

In our implementation, we seed the Tabu search with the Min-Min allocation where all tasks are scheduled in P-state two (i.e., $P_2$). All subsequent long-hops are generated by assigning each task to a random node in a random P-state, considering the tasks in a fixed, arbitrary order. Within each node and P-state, the task is mapped to the core within the node that gives the minimum expected completion time (MECT). The purpose of long-hops is to jump to a new neighborhood. Recall that the solutions generated by the long hops must differ from all solutions in the tabu list by at least 50% for all task-to-node assignments. If the generated long-hop does not fall within a new, unique neighborhood, a new long-hop is generated.

We use three short hop sub-procedures: 1) *meet-robustness*, 2) *reassignment*, and 3) *swapping*. For all of these procedures, solution $A$ is "better" than solution $B$ if the expected energy consumption of $A$ is better (i.e., lower) and $A$ meets the robustness constraint, or if the expected makespan of $A$ is lower than the expected makespan of $B$ and the robustness of $B$ is $< 90\%$. Each of these procedures is described below.

In the *meet-robustness* procedure, we iteratively attempt to minimize makespan (which indirectly works toward meeting the robustness constraint). For each iteration, the core that has the largest finishing-time is selected, and then all tasks on that core are unmapped. Next, the Min-Min heuristic is performed on the unmapped tasks. If the core with the maximum finishing-time is the same from the last iteration, a random core is selected. The routine terminates when either the mapping meets the robustness constraint or the number of iterations meets $max_{SH}$.

The *reassignment* short hop mechanism iteratively attempts to move tasks to other nodes. For each iteration, a random task is selected for re-assignment to a different node. When a task is considered for re-assignment, the MECT core within that node is chosen. Each re-assignment that yields a better solution is kept and the routine terminates when the number of iterations reaches $max_{SH}$.

The *swapping* short hop mechanism iteratively attempts to find the local best allocation by swapping tasks. For each iteration, two random tasks are selected for swapping. The core assignments of the first task and second task are swapped, and random P-states are chosen for each. The swap is kept if the solution improves. As with the previous short-hop procedures, the routine terminates when the number of iterations meets $max_{SH}$.

Pseudo-code for the entire Tabu heuristic is given in Algorithm 1.

Tabu does the *reassignment* and *swapping* short-hops twice for the following reason: Because each long-hop is randomized, solutions generated by long-hops generally have very low probabilities of completing by the system deadline. The Tabu heuristic first attempts a local search around the original long-hop by running a number of *reassignment* and *swapping* short-hops. At this point, the modified solution is still not likely to meet the robustness constraint. Therefore, the heuristic then runs the *meet-robustness* short-hop procedure, which is very likely to make the solution meet the robustness constraint. However, the *meet-robustness* short-hop procedure does not directly attempt to minimize energy consumption. To remedy this, the *reassignment* and *swapping* short-hop procedures are run again to try and reduce energy consumption.

**Algorithm 1** Pseudo-code for the Tabu heuristic

```
1: best ⇐ ∅
2: $n_{LH} \Leftarrow 0$
3: while $n_{LH} < max_{LH}$ do
4:     solution ⇐ next long-hop
5:     solution ⇐ reassignment short hop
6:     solution ⇐ swapping short hop
7:     solution ⇐ meet-robustness short hop
8:     solution ⇐ reassignment short hop
9:     solution ⇐ swapping short hop
10:    if solution is better than best then
11:        best ⇐ solution
12:    end if
13:    $n_{LH} \Leftarrow n_{LH} + 1$
14: end while
```

## V. Lower Bound on Energy Consumption

We calculate a lower bound (LB) on expected energy consumption for each simulation trial to help evaluate the heuristics. The LB is based on two parts: (a) a LB on the expected energy consumed by the execution of all tasks, and (b) a LB on the expected energy consumed by overhead. Note that our LB does not take the robustness constraint into account.

The LB on energy consumed by tasks is found as follows. For each task, find the node/P-state combination that minimizes energy consumed by that single task, and sum over all tasks. Let $\boldsymbol{H_{ti\pi}}$ be the expected energy consumption of task $t$ on node $i$ (recall that all cores within a node are homogeneous) in P-state $\pi$ (not taking node efficiency into consideration). This quantity is then divided by the maximum node efficiency across all nodes. Let $\boldsymbol{\zeta^{min}_{tasks}}$ denote the LB on energy consumption for part (a):

$$\zeta^{min}_{tasks} = \frac{1}{\underset{\forall i}{max}(\epsilon_i)} \sum_{\forall t \in T} \underset{\forall i,\pi}{min}\,(H_{ti\pi}). \tag{8}$$

For part (b), assume that any task can execute on any node using its minimum expected execution time over all nodes. The sum of the minimum expected execution time values is the minimum total task computation time for all tasks, $\boldsymbol{F_{min}}$. This time must be divided among the available cores in the system, and it should be divided in such a way that the expected energy consumed by overhead is minimized.

Let $F_{some}$ denote the computation time on node $i$. The energy consumed by overhead within node $i$ is a function of node parameters and $F_{some}$. Node parameters cannot change, but we can decrease the computation time by parallelizing $F_{some}$ as much as possible over the available cores in node $i$. When we parallelize $F_{some}$ within node $i$, two effects occur. First, the energy consumption due to node overheads decreases, because the completion time of the node decreases from $F_{some}$ to $F_{some}/(n_i n_{ij})$. Second, the energy consumption due to multicore processor overheads either stays the same or decreases. If there is only one core per multicore processor, then the energy consumption due to multicore processor overheads will remain constant, because the completion time of each core decreases from $F_{some}$ to $F_{some}/(n_i)$, but the multicore processor overhead increases from $\omega_i^{MP}$ to $n_i\omega_i^{MP}$. If there is more than one core in each multicore processor of the node, the energy consumed by multicore processor overheads will decrease with decreasing $F_{some}$.

From the above, it should be easy to see that it is always best to fully parallelize computation time within a node to minimize the energy consumed due to overhead. If we can perfectly parallelize computation time within a node $i$, then the completion time of node $i$ ($F_i$) will be minimized. Also, because we have perfectly parallelized the computation time, $F_i = F_{ij}$ for all multicore processors within node $i$. Thus, we can express the overhead power consumption for any node $i$, denoted $\boldsymbol{\omega_i^{total}}$, as

$$\omega_i^{total} = \frac{\omega_i^{node}}{\epsilon_i} + \frac{n_i\omega_i^{MP}}{\epsilon_i}. \tag{9}$$

If we let $F^{(i)}_{total}$ denote the sum of the expected execution time values of the tasks assigned to node $i$, then we can express the total overhead energy consumption for node $i$, denoted $\boldsymbol{\zeta^{(i)}_{overhead}}$, as

$$\zeta^{(i)}_{overhead} = \frac{F^{(i)}_{total}}{n_i n_{ij}} \times \omega_i^{total}. \tag{10}$$

We can use the above equations with our earlier assumption that a task can execute on any node in its minimum execution time to easily bound the energy consumption due to overheads. At least $F_{min}$ time must be spent computing, and the computation can be spread over any number and configuration of nodes. However, if there is a node with the smallest value of $\omega_i^{total}/n_i n_{ij}$ among all nodes, then executing the entire workload on that node will result in the minimum energy consumption from overheads. This is because, although the workload could be partitioned among $q$ other nodes, executing on those $q$ nodes for a duration of $F_{min}/q$ will have a higher energy consumption than executing on the minimum overhead node for $F_{min}$ and having no other overhead energy consumption.

From the above, we can derive the LB on energy consumption for part (b), denoted $\boldsymbol{\zeta^{min}_{overhead}}$, as

$$\begin{aligned}\zeta^{min}_{overhead} = \sum_{t_i \in T} & \underset{\forall i\pi}{min}\,(ETC\{t_i,\pi\}) \times \\ & \underset{\forall i}{min}\left(\frac{\omega_i^{total}}{n_i n_{ij}}\right).\end{aligned} \tag{11}$$

Finally, the LB on expected energy consumption, denoted $\boldsymbol{\zeta^{min}}$ is

$$\zeta^{min} = \zeta^{min}_{tasks} + \zeta^{min}_{overhead}. \tag{12}$$

## VI. Simulation Setup

To test the heuristics, 50 unique simulation trials were generated. For all trials, the number of tasks was set to 4000. Between simulation trials, only the means and variances of the task execution time distributions for each P-state in each node differed. All other parameters were held constant.

The cluster consisted of 25 compute nodes ($N$), where each node incorporated from one to four multicore processors ($n_i$), and each multicore processor included one to four cores ($n_{ij}$). Also, the makespan constraint, $\delta$, was set to 210 time units.

The mean and variance values of the task execution time distributions for each P-state in each node (ETC $\{t_{ijk}, \text{pstate}(t_{ijk})\}$ and V $\{t_{ijk}, \text{pstate}(t_{ijk})\}$, respectively) were generated using the Coefficient of Variation (COV) method from [4] and a scaling procedure. More specifically, the mean values for all task execution times on all nodes in the lowest P-state were generated using the COV method, and the values were then scaled by a random number between 0.9 and 0.95 to obtain the mean values for all task execution times on all nodes in the second P-state. This procedure was then repeated for the remaining P-states to generate the remaining mean values. The variance values were generated in a similar manner, where the COV method was used to generate the variance values for all task execution times on all nodes in the highest P-state, and a random scaling factor between 1.05 and 1.10 was used to scale the variance values for all task execution times on all nodes in the remaining P-states. In this way, both the mean and variance of each task execution time on a given node decreases with increasing P-state.

The power consumption values for each node and P-state ($\rho_i^{(\pi)}$) and efficiency values for each node ($\epsilon_i$) were generated once. Node efficiency values for each of the 25 nodes were chosen by sampling a uniform distribution with values from 0.75 to 0.90. Power consumption values for each node in each P-state were chosen in two phases. In the first phase, the power consumption value for each node in its lowest P-state was chosen by sampling a normal distribution with an empirically-determined mean and variance. The power consumption values for each node in its highest P-state were also chosen by sampling a normal distribution with an empirically-determined mean and variance. In the second phase, the power consumption values for each node and the remaining P-states were obtained by interpolating the values along a quadratic curve, that forced all power consumption curves to follow a quadratic shape.

Finally, the node and multicore processor overhead values for each node ($\omega_i^{node}$ and $\omega_i^{MP}$, respectively) were generated by sampling a uniform distribution. The bounds on the uniform distribution were carefully selected such that the total average overhead energy comprised approximately 30% of the total energy consumed by the system. In our trials, these values ranged from 20-30W for $\omega_i^{node}$, and 10-20W for $\omega_i^{MP}$.

## VII. Results

A box and whiskers plot of the expected energy consumption is shown in Figure 3 for each of the heuristics. Data collected over the 50 simulation trials is summarized within the box and whiskers: the bounds of the box represent the first and third quartiles, the line in between them represents the median, and the whiskers represent the minimum/maximum values. The Min-Max heuristic is not shown because it exhibited the same general behavior as the Min-Min heuristic except with slightly worse performance. For comparison, the lower bound is shown as well to provide insight into how well each heuristic performed overall. In addition, the expected makespan and robustness values are presented in Figures 4 and 5 respectively.

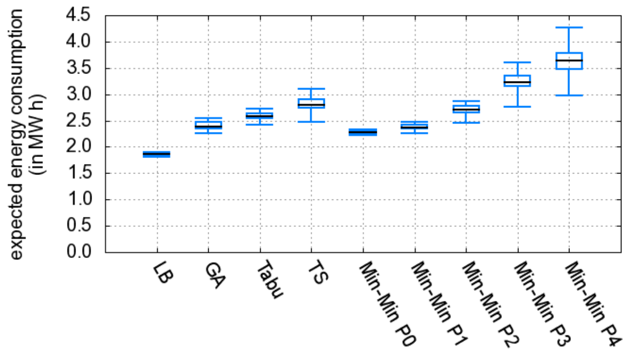

Fig. 3. Expected energy consumption for each heuristic, with the Min-Min heuristic run in each P-state.

First, consider the behavior of the Min-Min heuristic in the various different P-states. As expected, lower P-states result in lower total expected energy and higher expected makespan. However, the reduction in energy for task execution in lower P-states is offset by the increased energy consumption due to node and multicore processor overhead. That is, because lower P-states have longer execution times, this overhead is incurred over a longer time period so that there are diminishing returns in terms of reducing total energy consumption. In P-state one, Min-Min is unable to satisfy the robustness constraint of 90%, as can be seen in Figure 5. However, P-state two is not only able to meet the robustness constraint but also achieves a relatively low level of expected energy consumption. However, there is still room for improvement because the robustness is not at its constraint. Clearly, because having all of the tasks executing in P-state one violates the robustness constraint, a combination of different P-states should be able to improve expected energy consumption while driving the robustness to its constraint.

In terms of non-greedy heuristics, the tree-search algorithm actually performed slightly worse than the Min-Min in P-state two. It also did not result in allocations that just met the robustness constraint. One reason for this may be due to the limited number of active nodes. That is, in this implementation of tree search, the maximum number of active nodes was limited to 40, because for values any higher than this, the heuristic overruns the available memory. However, the performance of tree search may improve with a larger active node count. In addition, considering more levels in the tree simultaneously and/or modifying the criteria for which nodes to explore may also improve its performance.

Tabu is slightly better than Min-Min because its best solutions were typically found after it had completed all of the short hop procedures on the initial Min-Min seed. The resulting solution was on average approximately 5% better than the Min-Min P-state two solution. Tabu achieved this result by driving the robustness to its constraint. Unfortunately, the pure random solutions found by the long hops within Tabu generally failed to meet the robustness constraint. Therefore, all of the short hops were used to repair the long hop solutions to meet the robustness constraint rather than for minimizing expected energy consumption.

The GA heuristic was the clear winner, in that it achieved the lowest expected energy consumption while exactly meeting the robustness constraint. It is interesting to note that the median expected energy consumption is comparable to that of the Min-Min in P-state one despite the fact that the median robustness of Min-Min in P-state one is less than 4% while the median robustness of the GA solution is 90%. The GA result is still significantly higher than the lower bound, but this may be due more to the looseness of the bound than the quality of the GA solution. This conjecture is reinforced by the result of the P-state zero Min-Min solution that failed to meet the robustness constraint but provided the lowest expected energy consumption.

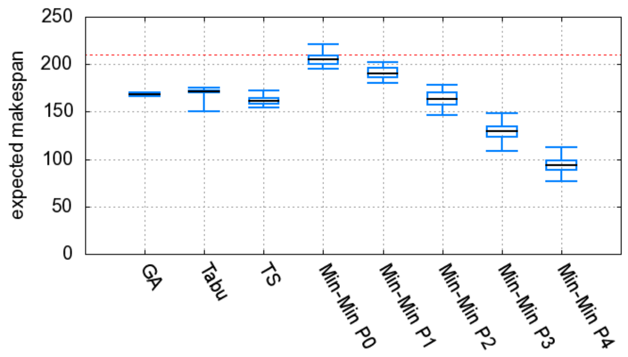

Fig. 4. Expected makespans for each heuristic, with the Min-Min heuristic run in each P-state.

## VIII. Related Work

The problem of mapping a set of independent tasks while trying to minimize energy consumption has been previously investigated in [8]. The goal in [8] was to statically allocate tasks to a multiprocessor system and minimize energy consumption. However, in contrast to our paper, there was no constraint on, or optimization of, the makespan of the set of tasks executed. Also, execution times were modeled using expected values, while we model task execution times as random variables.

In [20], a method was introduced for allocating dependent tasks onto multiple embedded processing elements that leveraged variable dynamic voltage scaling (**DVS**). The research used a pair of nested genetic algorithms to determine a schedule for multiple task graphs and optimize the system energy consumption. However, unlike our system, task execution times were known exactly. Further, a variable-voltage DVS system was assumed, so that *any* voltage level could be provided to tweak task execution times, as opposed to being limited to discrete P-states, specified in the ACPI standard [2] that we use. Additionally, the work in [20] focused on decreasing energy consumption by using DVS to fill slack in precedence-constrained schedules while not violating individual task deadlines, whereas our research focuses on scheduling a large set of tasks to optimize energy consumption under a makespan constraint, and includes overhead power consumption.

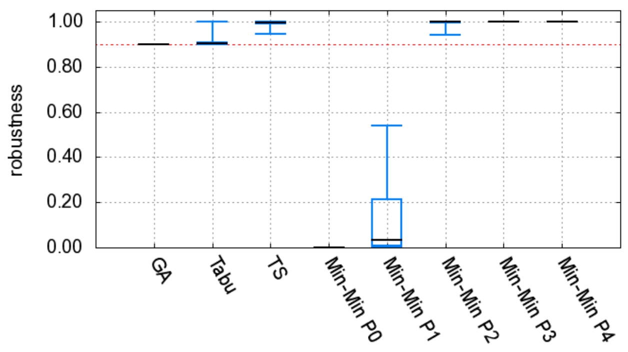

Fig. 5. Robustnesses for each heuristic, with the Min-Min heuristic run in each P-state.

Similar to [20], the environment in [28] is that of dependent tasks executing across multiple homogeneous processors. Task dependencies are modeled as direct acyclic graphs (DAGs) and the goal of that research is to minimize the total energy consumed by all tasks, employing DVFS to help achieve this goal. In [28], a two-phase framework is developed to schedule the set of tasks onto the processors. However, [28] does not place any constraint on the execution time of the set of tasks being executed. Also, the processors in [28] are homogeneous, unlike in this study where our model possesses multiple heterogeneous compute nodes.

In [14], a point is made that when considering minimization of energy with a collection of tasks, there is a trade-off between energy consumption and makespan. In that research, an objective function is proposed in an attempt to balance energy consumption and makespan. However, this is different from our research in that in [14] no constraint is placed on energy consumption or makespan. Furthermore, [14] does not address issues of uncertainty or robustness.

## IX. Conclusions

In this research, we consider the problem of robust resource allocation in a heterogeneous cluster. The goal of the resource allocation heuristics is to minimize energy using DVFS for executing a set of tasks while maintaining a quantifiable probabilistic guarantee on completing the tasks by a common deadline. We designed a model and measure for stochastic robustness that quantifies the reliability of an allocation to complete by a given deadline, and used the measure in three novel resource allocation heuristics. Finally, a lower bound on energy consumption was designed to help gauge the performance of each heuristic.

Our initial findings indicates the promise of the GA heuristic. For future work, we would like to integrate a memory hierarchy model into our environment. We also would like to model task execution times as general pdfs instead of normal distributions.

## REFERENCES

[1] SPECpower_ssj2008. [Online]. Available: http://www.spec.org/power_ssj2008/results/power_ssj2008.html

[2] *Advanced Configuration and Power Interface Specification*, Std. 4.0a, 2010.

[3] S. Ali, A. A. Maciejewski, H. J. Siegel, and J.-K. Kim, "Measuring the robustness of a resource allocation," *IEEE Trans. Parallel and Distributed Systems*, vol. 15, no. 7, pp. 630–641, Jul. 2004.

[4] S. Ali, H. J. Siegel, M. Maheswaran, and D. Hensgen, "Representing task and machine heterogeneities for heterogeneous computing systems," *Tamkang J. of Science and Engineering, Special 50th Anniversary Issue*, vol. 3, no. 3, pp. 195–207, Nov. 2000.

[5] 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: Chapman & Hall/CRC Press, 2008, pp. 41–1–41–30.

[6] L. A. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan & Claypool Publishers, 2009.

[7] 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. of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, Jun. 2001.

[8] P.-C. Chang, I.-W. Wu, J.-J. Shann, and C.-P. Chung, "Etahm: An energy-aware task allocation algorithm for heterogeneous multiprocessor," *45th Design Automation Conf. (DAC '08)*, pp. 776–779, 2008.

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

[10] S. Hartmann, "A self-adapting genetic algorithm for project scheduling under resource constraints," *Naval Research Logistics*, vol. 49, no. 5, pp. 433–448, 2002.

[11] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on non-identical processors," *J. of the ACM*, vol. 24, no. 2, pp. 280–289, Apr. 1977.

[12] A. Kumar and R. Shorey, "Performance analysis and scheduling of stochastic fork-join jobs in a multicomputer system," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 10, Oct. 1993.

[13] Y. M. Lam, J. G. F. Coutinho, W. Luk, and P. H. W. Leong, "Integrated hardware/software codesign for heterogeneous computing systems," *4th Southern Conf. on Programmable Logic 2008*, pp. 217–220, 2008.

[14] Y. C. Lee and A. Y. Zomaya, "Minimizing energy consumption for precedence-constrained applications using dynamic voltage scaling," *9th IEEE/ACM Int'l Symposium on Cluster Computing and the Grid*, pp. 92–99, 2009.

[15] 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. of Parallel and Distributed Computing*, vol. 44, no. 1, pp. 35–52, Jul. 1997.

[16] 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. of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 107–121, Nov. 1999.

[17] D. Meisner, B. T. Gold, and T. F. Wenisch, "PowerNap: Eliminating server idle power," *SIGPLAN Not.*, vol. 44, no. 3, pp. 205–216, 2009.

[18] Z. Michalewicz and D. B. Fogel, Eds., *How to Solve It: Modern Heuristics*. New York, NY: Springer-Verlag, 2000.

[19] A. M. Rahmani and M. A. Vahedi, "A novel task scheduling in multiprocessor systems with genetic algorithm by using elitism stepping method," *Int'l J. of Computer Theory and Engineering*, vol. 1, no. 1, pp. 1–6, 2009.

[20] M. Schmitz, B. Al-Hashimi, and P. Eles, "Energy-efficient mapping and scheduling for DVS enabled distributed embedded systems," *2002 Design, Automation and Test in Europe Conf. and Exhibition (DATE '02)*, pp. 514–521, 2002.

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

[22] J. Smith, H. J. Siegel, and A. A. Maciejewski, "Robust resource allocation in heterogeneous parallel and distributed computing systems," in *Wiley Encyclopedia of Computer Engineering*, B. W. Wah, Ed. John Wiley & Sons, Hoboken, NJ, 2009, vol. 4, pp. 2461–2470.

[23] J. Smith, E. K. P. Chong, A. A. Maciejewski, and H. J. Siegel, "Stochastic-based robust dynamic resource allocation in a heterogeneous computing system," *38th Int'l Conference on Parallel Processing (ICPP-2009)*, Sep. 2009.

[24] S. L. Suchita Upadhyaya, "Task allocation in distributed computing vs distributed database systems: A comparative study," *Int'l J. of Computer Science and Network Security*, vol. 8, no. 3, pp. 338–346, 2008.

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

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

[27] M. Wu, W. Shu, and H. Zhang, "Segmented min-min: A static mapping algorithm for meta-tasks on heterogeneous computing systems," *9th IEEE Heterogeneous Computing Workshop*, pp. 375–385, Mar. 2000.

[28] Y. Zhang, X. S. Hu, and D. Z. Chen, "Task scheduling and voltage selection for energy minimization," *45th Design Automation Conf. (DAC '08)*, pp. 183–188, 2002.