Dissertation

Resource Management for Heterogeneous Computing Systems: Utility Maximization, Energy-Aware Scheduling, and Multi-Objective Optimization

Submitted by

Ryan Friese

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2015

Doctoral Committee:

Advisor: H. J. Siegel
Co-Advisor: Anthony A. Maciejewski

Sudeep Pasricha
Gregory A. Koenig
Patrick J. Burns

Abstract

Resource Management for Heterogeneous Computing Systems: Utility Maximization, Energy-Aware Scheduling, and Multi-Objective Optimization

As high performance heterogeneous computing systems continually become faster, the operating cost to run these systems has increased. A significant portion of the operating costs can be attributed to the amount of energy required for these systems to operate. To reduce these costs it is important for system administrators to operate these systems in an energy efficient manner. Additionally, it is important to be able to measure the performance of a given system so that the impacts of operating at different levels of energy efficiency can be analyzed. The goal of this research is to examine how energy and system performance interact with each other for a variety of environments. One part of this study considers a computing system and its corresponding workload based on the expectations for future environments of Department of Energy and Department of Defense interest. Numerous Heuristics are presented that maximize a performance metric created using utility functions. Additional heuristics and energy filtering techniques have been designed for a computing system that has the goal of maximizing the total utility earned while being subject to an energy constraint. A framework has been established to analyze the trade-offs between performance (utility earned) and energy consumption. Stochastic models are used to create "fuzzy" Pareto fronts to analyze the variability of solutions along the Pareto front when uncertainties in execution time and power consumption are present within a system. In addition to using utility earned as a measure of system performance, system makespan has also been studied. Finally, a framework has been developed that enables the investigation of the effects of P-states and memory interference on energy consumption and system performance.

Acknowledgements

This dissertation would not have been possible without the help, guidance, and support of many people. A special thanks to my advisors: Dr. Howard Jay Siegel and Dr. Anthony A. Maciejewski for the tremendous amount of time and energy they spent guiding, critiquing, teaching, and mentoring me over the past 6 years. I cannot thank them enough. I also thank the rest of my graduate committee: Dr. Sudeep Pasricha, Dr. Greg Koenig, and Dr. Patrick Burns for the input and constructive feedback. Many parts of this work simply would not have been possible without their guidance.

To our collaborators at Oak Ridge National Lab and the Department of Defense: Chris Groer, Marcia Hilton, Gene Okonski, Steve Poole, Sarah Powers, Jendra Rambharos, Mike Wright, and Neena Imam, I thank you for your insight, willingness, and patience over the past four years.

I thank Dr. Bhavesh Khemka for being a great friend and great teammate, I will always appreciate the many hours we have spent working together, laughing, and banging our heads against the wall trying to solve numerous problems.

To all the members of the CSU robust computing research group that I have worked with over the years: Dr. Abdulla Al-Qawasmeh, Dr. Mohsen Amini, Jonathan Apodaca, Dr. Luis D. Briceño, Tyler Brinks, Daniel Dauwe, Dr. Tim Hansen, Eric Jonardi, Dr. Paul Maxwell, Curt Oliver, Mark Oxley, Dr. Greg Pfister, Dr. Jay Smith, Dr. Kyle Tarplee, and Dalton Young, thank you for your valuable comments on my research, papers, and presentations.

I thank my family: Kurt and Lydia Friese, and Natasha, Brandon, Nolen, and Lila Henderson for their love and care throughout my life. Their support and commitment to my success helped get me to where I am today.

Last and far from least, I thank Jamie Roberts. She deserves considerable recognition for supporting me for the last six years. She has always been there for me, through all the papers and presentations, all the late nights and early mornings, and all the joys and frustrations. I thank her for her unshakable belief and confidence in me, I will never be able to fully express the gratitude I have for her.

This research was made possible thanks to the funding and support of the National Science Foundation Graduate Research Fellowship, the Colorado State University George T. Abell Endowment, the National Science Foundation under grant numbers CNS-0905339, CCF-1252500, and CCF-1302693, and the Oak Ridge National Laboratory and their Extreme Scale Systems Center under subcontract numbers 4000094858 and 4000108022. Parts of this research used the CSU ISTeC HPC System supported by National Science Foundation under grant number CNS-0923386. Parts of this research used systems donated by Hewlett Packard (HP). Any opinions, findings, and conclusions or recommendations are those of the author and do not necessarily reflect the views of the National Science Foundation.

This dissertation is typset in LaTeX using a document class designed by Leif Anderson.

DEDICATION

To my family whose love, caring, and wisdom are beyond compare

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction and Overview

Achieving high performance computing (HPC) is critical to a number of important design problems of national interest, including disastrous weather prediction, drug design, manufacturing, etc. The goal of this research is to analyze and design energy-aware resource allocations to maximize the performance and minimize the energy consumption of high performance heterogeneous computing systems. Two major impediments to the growth of HPC are energy consumption and the effective use of heterogeneous resources, consisting of different machines, interconnecting communications, etc. About 60% of data centers state that the available power and cooling capacity are key factors limiting data center growth [1]. In heterogeneous systems, machines perform differently for a given task, thus as the complexity of the system increases, so does the difficulty in assigning tasks in a performance and energy efficient manner. This research addresses both concerns by developing mathematical and algorithmic tools to quantify the performance of a system, perform energy-aware resource allocation, and utilize multi-objective optimization to analyze the trade-offs between performance and energy consumption. These tools are used to create efficient and robust resource allocations for a given environment, where there are uncertainties in workload, computing platform, and system environment parameters. This research significantly contributes to the current and future national needs for energy-aware computing systems.

Chapter 2 models an oversubscribed heterogeneous computing system where tasks arrive dynamically and a scheduler maps the tasks to machines for execution. The environment and workloads are based on those being investigated by the Extreme Scale Systems Center at Oak Ridge National Laboratory. Utility functions that are designed based on specifications

from the system owner and users are used to create a metric for the performance of resource allocation heuristics. Each task has a time-varying utility (importance) that the enterprise will earn based on when the task successfully completes execution. Multiple heuristics are designed, and include a technique to drop low utility-earning tasks, to maximize the total utility that can be earned by completing tasks. The heuristics are evaluated using simulation experiments with two levels of oversubscription. The results show the benefit of having fast heuristics that account for the importance of a task and the heterogeneity of the environment when making allocation decisions in an oversubscribed environment. The ability to drop low utility-earning tasks allow the heuristics to tolerate the high oversubscription as well as earn significant utility.

The need for greater performance in high performance computing systems combined with rising costs of electricity to power these systems motivates the need for energy-efficient resource management and is examined in Chapter 3. Driven by the requirements of the Extreme Scale Systems Center at Oak Ridge National Laboratory, the problem of scheduling dynamically-arriving tasks to machines in an oversubscribed and energy-constrained heterogeneous distributed computing environment is addressed. The goal is to maximize total "utility" earned by the system, where the utility of a task is defined by a monotonically-decreasing function that represents the value of completing that task at different times. To address this problem, four energy-aware resource allocation heuristics have been designed and their performance is compared to heuristics from the literature. For the given energy-constrained environment, an energy filtering technique is also designed that helps some heuristics regulate their energy consumption by allowing tasks to only consume up to an estimated fair-share of energy. Extensive sensitivity analyses of the heuristics in environments with different levels of heterogeneity show that heuristics with the ability to balance both

energy consumption and utility exhibit the best performance because they save energy for use by future tasks.

Chapter 4 continues to examine the need for energy-efficiency in high performance computing, especially with systems approaching exascale levels. The Extreme Scale Systems Center at Oak Ridge National Laboratory faces a need for resource management techniques that maximize the performance of the system while satisfying an energy budget. The performance of the system is measured as the total "utility" earned from completing tasks. Utility is represented as a time-varying importance of a task. This chapter performs an in-depth examination into the energy-constrained utility maximization problem by comparing the performance of resource management techniques in two different task management environments: queued and polled. In one environment, tasks are queued for execution on the different machines and certain tasks are not allowed to be re-scheduled. In the other environment, machines are polled at regular intervals and each idle machine is only assigned one task.

Before systems such as supercomputers, servers, and datacenters can begin operating in an energy-efficient manner, the energy consumption and performance characteristics of the system must be analyzed. In Chapter 5, an analysis framework is provided that will allow a system administrator to investigate the trade-offs between system energy consumption and utility earned by a system (as a measure of system performance). These trade-offs are modeled as a bi-objective resource allocation problem. A popular multi-objective genetic algorithm (NSGA-II) is used to construct Pareto fronts to illustrate how different resource allocations can cause a system to consume significantly different amounts of energy and earn different amounts of utility. The analysis framework is demonstrated using real data collected from online benchmarks, and further a method to create larger data sets that exhibit similar

heterogeneity characteristics to real data sets is provided. This analysis framework can provide system administrators with insight to make intelligent scheduling decisions based on the energy and utility needs of their systems.

To help facilitate a transition to energy-efficient computing, the trade-offs between system performance and system energy consumption must be analyzed and understood. Chapter 6 models minimizing energy consumption and maximizing system performance as a bi-objective optimization problem and utilizes the framework introduced in Chapter 5 to construct Pareto fronts (via simulation) consisting of Pareto-efficient resource allocations. These Pareto fronts illustrate the trade-offs between minimizing energy consumed and minimizing the makespan of a system. Two studies are presented in this chapter. In the first study, different solutions from within the Pareto fronts are analyzed to further understand the relationships between energy consumption and makespan. In the second study, three environments, based on data collected from real machines and real applications, are analyzed and it is shown that by utilizing different resource allocations, a system administrator would be able to significantly change the performance and energy consumption of a given system. This analysis provides the system administrator with the means to examine these trade-offs to help make intelligent decisions regarding the scheduling and composition of their systems. Finally, this study shows how system administrators can perform "what-if" analyses to evaluate the effects of adding or removing machines from their high performance computing systems.

Chapter 7 continues to analyze the trade-offs between performance and energy consumption to make informed decisions for a given compute system and extends the framework presented in Chapters 5 and 6 to account for stochastic execution times and energy consumption values. In high performance computing systems, tasks typically have different

execution times depending on the data sets they use. These variations in the execution time of the tasks can be modeled by stochastic distributions. In this chapter, a new approach for performing stochastic multi-objective optimization comparing utility earned vs. energy consumed is presented. Two different objective function sets used by the NSGA-II are detailed and analyzed, one is based on expected values, and the other is probabilistic-based.

Multi-core processors are prevalent in today's computing world, and they are the "building blocks" of high performance computing systems. Chapter 8 analyzes interference trends when co-running multiple applications possessing varying degrees of memory intensity on multi-core processors. Numerous tests are conducted using PARSEC benchmark applications to explore energy consumption, execution times, and main memory accesses when interfering applications share last-level cache. This chapter also explores how co-running applications are impacted when the processor frequency is modified using dynamic voltage and frequency scaling (DVFS). A portable and lightweight testing framework is presented and results are shown for experiments conducted on an Intel i7 quad-core system. It is shown that the degree of degradation due to co-location interference on execution time is highly dependent on the types and number of applications co-located on cores that share the last-level cache.

A description of possible future directions for the studies presented in this document are in Chapter 9.

CHAPTER 2

Utility Maximization in Resource Management[1]

2.1. Introduction

A utility function for a task describes the value of completing the execution of the task at a specific time [3–7]. Utility functions capture the time-varying importance of a task to both the user and the enterprise as a whole. In this work, the value of completing a task decays over time and so monotonically-decreasing utility functions are modeled. The design of utility functions needs to be flexible to capture the importance of tasks within a diverse user base. In practice, utility functions may be created through a collaboration between the user and the owner of the computing system. Dynamic resource management techniques are designed to maximize the total utility that can be earned by completing tasks in an oversubscribed heterogeneous distributed environment. Oversubscribed means that the workload is large enough that the total desired work exceeds the capacity of the system in steady state operation, i.e., over an extended period. A heterogeneous environment means that the execution time of each task may vary across the suite of machines. This type of computing environment and the workload of tasks that arrive dynamically is modeled. A scheduler makes resource allocation decisions to map (assign) the incoming tasks to the machines. The total utility earned from all completed tasks captures how much useful work was done and how timely that information was to the user. The system characteristics and the workload parameters are based on environments being investigated by the Extreme Scale

[1]This work was done jointly with Ph.D. student Bhavesh Khemka. The full list of co-authors is at [2]. This work was supported by Oak Ridge National Laboratory and their Extreme Scale Systems Center under subcontract numbers 4000094858 and 4000108022, and by a National Science Foundation Graduate Research Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research also used resources of the Extreme Scale Systems Center at Oak Ridge National Laboratory as well as the CSU ISTeC Cray System supported by NSF Grant CNS-0923386.

Systems Center (ESSC) at Oak Ridge National Laboratory (ORNL). The ESSC is part of a collaborative effort between the Department of Energy (DOE) and the Department of Defense (DoD) to perform research and deliver tools, software, and technologies that can be integrated, deployed, and used in both DOE and DoD environments.

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

Two forms of dynamic heuristics are used for the resource allocation decisions. Immediate-mode heuristics schedule only the incoming task and do not have the opportunity to re-map tasks in machine queues (e.g., [8–10]). Batch-mode heuristics consider a set of tasks and have the ability to re-map tasks that are enqueued and waiting to execute (e.g., [8, 9]). Seven immediate-mode and five batch-mode heuristics were created, and their performance is analyzed using simulation experiments. To examine the effect of oversubscription on the performance of the heuristics, two levels of oversubscription were simulated. The effects of heuristic variations, such as dropping tasks and altering the mapping decision frequency for the batch-scheduler was also studied.

The contributions of this chapter are: (a) a model of the planned DOE/DoD oversubscribed heterogeneous high performance computing environment, (b) the design of a metric using utility functions, based on the three parameters of priority, urgency, and utility class,

to measure the performance of schedulers in an oversubscribed heterogeneous computing environment, (c) the design of twelve heuristics to perform the scheduling operations and their evaluation over a variety of environments, and (d) the exploration and the analysis of heuristic variations, such as dropping tasks and varying the number of tasks scheduled at each batch-mode mapping event.

The remainder of the chapter is organized as follows. In Sec. 2.2, the system model is explained, including the method to design the utility functions (from the three parameters mentioned before), the characteristics of the workload, and the characteristics of the computing environment. The problem statement is formally given in Sec. 2.3, and the metric to compare the performance of resource allocation heuristics is introduced. In Sec. 2.4, the various heuristics that were designed and the method to drop tasks are described. This study is compared to other work from the literature in Sec. 2.5. The design of the simulation experiments is explored in Sec. 2.6. In Sec. 2.7, the simulation results are presented and analyzed. Finally, the chapter concludes in Sec. 2.8.

2.2. System Model

2.2.1. Utility Functions.

2.2.1.1. *Overview.* In this study, it is assumed that an enterprise computing system earns a certain amount of utility for completing each task. The amount of utility earned depends on the task and the time at which the task was completed relative to the time it arrived, and reflects its importance to the system. Utility functions are used to model the time-varying benefit of completing the execution of a task. The utility functions modeled are monotonically decreasing. This implies that if a task takes longer to complete, it cannot earn higher utility. There may be use cases for non-monotonically-decreasing utility functions,

but they are not considered here. A flexible utility function for a task is defined by three parameters: priority, urgency, and utility class. The goal is to use a small set of parameters that the users understand and enables the users to obtain the desired utility curve. By using a combination of these parameters one can create a variety of shapes for the utility functions. These parameters were designed based on the needs of the ESSC at ORNL. It is expected that these parameters will be set by the customer (submitting the job) in collaboration with the system owner and the overall system administration policies.

2.2.1.2. *Parameters.*

Priority. Priority represents the importance of the task to the organization. It sets the maximum value of a utility function. As the functions are monotonically decreasing, this is equivalent to the starting value of the utility function. Let $\pi(p)$ be the maximum utility of tasks belonging to priority level p, where $p \in \{critical, high, medium, low\}$. Each of these priority levels has a fixed value of maximum utility associated with it. Fig. 2.1a shows utility functions with different levels of priority for a fixed level of urgency (defined below). As shown in Fig. 2.1a, a task's utility does not begin to decay as soon as it arrives, because this would make the maximum utility value of a task unachievable (i.e., the task needs non-zero time to execute). In Sec. 2.6.2, it is describe how the length of this interval is determined.

Urgency. The urgency of a task models the rate of decay of the utility of that task over time. It affects the "shape" of the utility function. Tasks that are more urgent will have their utility values decrease at a faster rate than less urgent tasks. In this study, the decay of utilities are modeled as exponentials (other functions may be used). Let $\rho(r)$ be the exponential decay rate of tasks belonging to urgency level r, where $r \in \{extreme, high, medium, low\}$. Fig. 2.1b illustrates utility functions with different urgency levels for a fixed priority level.

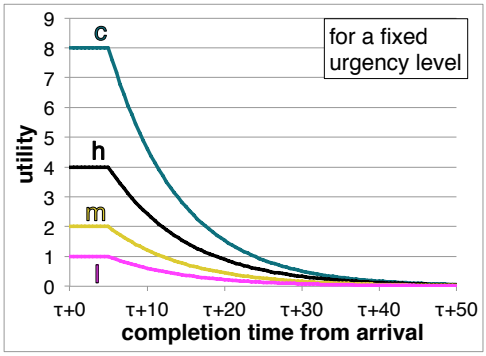
The urgency level of a task along with the task's average execution time control the duration for which the starting utility value of a task does not decay (see Sec. 2.6.2).

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

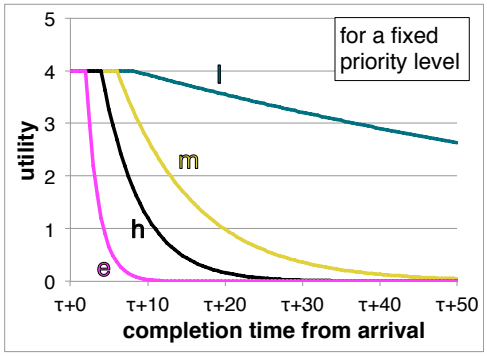
The first element within a utility class is the set of time intervals that partition the time axis of the utility function (except the end time of the first interval). Let $t(k, c)$ be the start time of the k^{th} interval relative to the arrival time of a task belonging to utility class c.

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

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



(A)



(B)

FIGURE 2.1. (a) Four utility functions with different priority levels and a fixed urgency level showing the decay in utility for a task after its arrival time τ. The curves labeled "c," "h," "m," and "l" are the curves with *critical*, *high*, *medium*, and *low* priorities, respectively. (b) Four utility functions with different urgency levels and a fixed priority level showing the decay in utility for a task after its arrival time τ. The curves labeled "e," "h," "m," and "l" are the curves with *extreme*, *high*, *medium*, and *low* urgency levels, respectively. The length of time for which the starting utility value of a task persists (does not decay) is shorter for more urgent tasks.

Fig. 2.2 shows a utility function (at a fixed priority and urgency level) partitioned into separate intervals, each with its own rate of decay and starting utility value. The last interval shows that the utility drops to zero as time tends to infinity.

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

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

2.2.2. MODEL OF ENVIRONMENT. Tasks with similar computational requirements are grouped into task types and machines with similar performance capabilities into machine types. A heterogeneous environment is modeled, where the execution times of different task types may vary across the different machine types. It is assumed an Estimated Time to

Compute (ETC) matrix is given, where $ETC(i, j)$ is the estimated time to compute a task of type i on a machine of type j. This is a common assumption in the resource management literature [11–16]. For simulation purposes, a synthetic workload as described in Sec. 2.6.3 is used, but in practice, one could use historical data to obtain such information [13, 15]. Special-purpose and general-purpose machines are modeled. The special-purpose machine types have the ability to execute certain task types much faster than the general-purpose machine types, but may be incapable of executing other task types. Further details are in Sec. 2.6.3.

This study models a dynamic environment where tasks arrive throughout a 24 hour period. The scheduler does not know the arrival time, utility function, or type of each task until the task arrives. The system is composed of dedicated compute resources with a workload large enough to create an oversubscribed environment. It is assumed that the tasks in the workload are independent (no inter-task communication is required) and serial (each task executes on a single machine). For scheduling purposes the pre-emption of tasks is not considered. However, tasks are allowed to be dropped prior to execution (see Sec. 2.4.4).

2.3. Problem Statement

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

calculated using Equation 1 to obtain $U(p(i), r(i), c(i), t_{completion}(i))$. Let $\Omega(t_{end})$ be the set of tasks that have completed execution by time t_{end}. The goal of the resource management procedures is to maximize the total utility that can be earned by the system over the 24 hour period and is computed using:

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

2.4. Resource Management Policies

2.4.1. Overview. The scheduling problem, in general, has been shown to be NP-complete [17], and therefore it is common to use heuristics to solve this problem. Any time when a decision has to be made to assign a task to a machine a mapping event is called. The two types of dynamic heuristics (also known as online heuristics [18]), immediate-mode heuristics and batch-mode heuristics, differ in the method that a mapping event is triggered and in the set of tasks that can be scheduled during a mapping event.

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

In batch-mode heuristics, typically mapping events are triggered after fixed time intervals or a fixed number of task arrivals. If the previous mapping event has not completed execution, the trigger time of the next mapping event is delayed until the previous mapping event finishes execution. The task that is next in-line for execution on a machine queue is referred

to as a pending task. The part of the machine queues that do not include the executing and the pending tasks referred to as the virtual queues of the scheduler. Fig. 2.3 shows tasks waiting in the virtual queues of an example system with four machines. The batch-mode heuristics make mapping decisions for both the tasks that have arrived since the last mapping event and the tasks that are waiting in the virtual queues. This set of tasks is the called the mappable tasks. The batch-mode heuristics (unlike immediate-mode heuristics) have the capability to re-map tasks in the virtual queues of scheduler. The batch-mode heuristics do not re-map pending tasks so the machine does not become idle when its executing task completes. The simulation results in Sec. 2.7 show that the batch-mode heuristics have a significant advantage because they have more information available while making a mapping decision (as they consider a set of tasks). Furthermore they can alter those decisions in the future by remapping tasks when additional information becomes available. Five batch-mode heuristics were designed and evaluated.

2.4.2. Immediate-mode Heuristics.

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

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

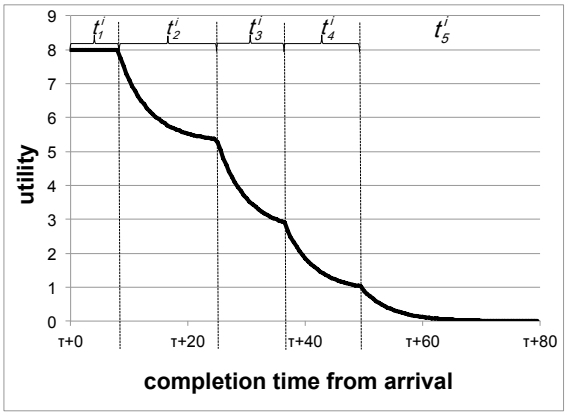


FIGURE 2.2. Utility function for a fixed priority level, urgency level, and utility class, showing the decay in utility for a task after its arrival time τ. The t^is represent the duration of the different intervals in the utility class of task i. The last interval extends to infinity.

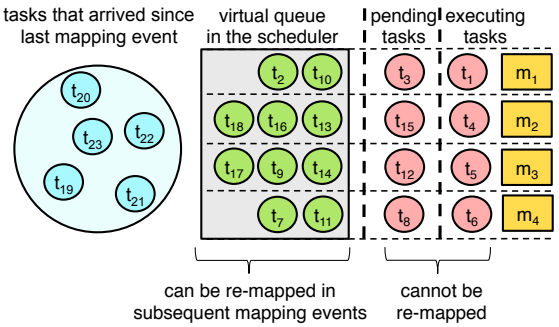


FIGURE 2.3. Machine queues of a sample system with four machines. The tasks in the executing and pending slots are not eligible to be re-mapped, whereas the tasks in the virtual queue section of the machine queues can be re-mapped. This only applies to the batch-mode heuristics.

event is assigned to the first machine (on which it can execute), the next incoming task is assigned to the next machine (on which it can execute), and so on.

2.4.2.2. *Smart Immdediate-mode Heuristics.* The next five heuristics are referred to as the smart immediate-mode heuristics. The results in Sec. 2.7 show that these heuristics perform better than the naive immediate-mode heuristics.

The Maximum Utility (Max Util) heuristic is based on the Minimum Completion Time heuristic from the literature [19, 20, 9, 21, 22]. The heuristic assigns a newly arrived task to the machine that would complete it soonest. Due to modeling monotonically-decreasing utility functions, the machine that completes the task the earliest is also the machine that earns the highest utility from the task. This heuristic accounts not only for the execution time of the task on machines, but also the ready-time of the machines.

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

Two heuristics were designed based on the Minimum Execution Time (MET) heuristic [19, 20, 9]. The Minimum Execution Time-Random (MET-Random) heuristic first finds the set of machines that belong to the machine type that can *execute* the newly-arrived task the fastest (ignoring machine ready time). Among those machines, it assigns the task to a random machine. The Minimum Execution Time-Max Util (MET-Max Util) heuristic also finds the set of machines belonging to the minimum *execution* time machine type for the newly arrived task, but picks the machine among them that minimizes *completion* time (which also maximizes utility).

The K-Best Types heuristic is based on the K-Percent Best heuristic, introduced in [9] and used in [23, 20, 24, 25]. The idea is to try combining the benefits of the MET heuristic and the Max Util heuristic. The K-Best Types heuristic first finds the K-best machine types that have the lowest *execution* times for the current task. Among the machines of these machine types, it then picks the machine that minimizes *completion* time (which also maximizes utility). By using different values of K, one can control the extent to which the heuristic is biased towards MET-Max Util or Max Util. The best value of K was determined empirically.

2.4.3. Batch-mode Heuristics. The Min-Min Completion Time (Min-Min Comp) heuristic is based on the concept of the two-stage Min-Min heuristic that has been widely used (e.g., [20, 26, 24, 25, 27, 8, 9, 28, 21, 22, 29]). In the first stage, the heuristic independently finds for each mappable task the machine that can complete it the soonest. In

the second stage, the heuristic picks from all the task-machine pairs (of the first stage) the pair that has the earliest completion time. The heuristic assigns the task to that machine, removes that task from the set of mappable tasks, updates the ready-time of that machine, and repeats this process iteratively until all tasks are mapped. This batch-mode heuristic is computationally efficient because it explicitly does not perform any utility calculations.

The Sufferage heuristic concept introduced in [9] and used in, for example, [30, 31, 26, 27, 24, 8, 21], attempts to assign tasks to their maximum utility machine. Ties are broken in favor of the tasks that would "suffer" the most if they did not get their maximum utility machine. In the first stage, the heuristic calculates for each mappable task a sufferage value, i.e., the difference between the best and the second-best utility values that the task could possibly earn. In the second stage, tasks are assigned to their maximum utility machines. If multiple tasks request the same machine, then the task that has the highest sufferage value is assigned to that machine. Assigned tasks are removed from the mappable tasks set, ready-times of machines updated, and the process repeated until all tasks are mapped.

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

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

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

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

load. Different methods to drop tasks in the immediate-mode and the batch-mode heuristics are used.

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

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

In addition to the dropping operation, for the batch-mode heuristics, a technique to permute tasks that are at the head of the virtual queues of the machines was implemented, but this did not improve performance. This technique is described in App. A.

2.5. Related Work

Numerous studies have proposed heuristics to solve the problem of performing resource management in dynamic heterogeneous computing environments (e.g., [25, 24, 8, 9]). Few

of them, however, optimize for the total utility that the system earns. In a survey of utility function based resource management [4], the authors point out that in an oversubscribed system it is preferable to use utility accrual algorithms for performing scheduling decisions because these have the ability to pick and execute tasks that are more important to the system (earn high utility). Additional research explores developing a framework for measuring the productivity of supercomputers [32]. They propose a metric for productivity that is the ratio of the utility earned by completing a task to the cost of doing this operation. Possible shapes for the utility-time and the cost-time curves of a task are discussed. The authors also mention the possible interpretations of "utility" and "cost." Similar to this work, they consider only monotonically-decreasing utility-time functions. The work presented in this chapter enhances this by parameterizing the shape of the utility functions and designing resource management techniques to maximize the aggregate utility.

Value functions (similar to utility functions) are used in systems with processes running on symmetric, shared-memory multi-processors (SMP) with one to four processing elements [3]. Each process has a value function associated with it that specifies the value earned by the system depending on when it completes execution of that process. The scheduler can consider the arrival times of tasks to make current scheduling decisions. Moreover, the processes can be periodic. This is in contrast to the model presented in this chapter where the scheduler has no prior knowledge of the arrival time of the tasks. The paper presents two algorithms that make decisions based on value density (value divided by processing time) and shows that these algorithms perform better than scheduling algorithms that consider either only deadlines or only execution times (ignoring the utility earned). This is similar to some of heuristics presented here that use utility-per-time. Unlike this environment, they

consider homogeneous processing elements. Other systems using similar value functions have also been examined [33, 34].

Kim et al. define tasks with three soft deadlines [24]. The actual completion time of the task is compared to the soft deadlines to obtain a deadline factor. The deadline factor is multiplied with the priority of a task to calculate the actual "value" that is earned for completing the task. Dynamic heuristics are used to maximize the total value that can be earned by mapping the tasks to machines. Although tasks can have different priorities, the degradation curve for the value of a task is always a step-curve with the steps occurring at the soft deadlines. In the presented model, each task can have its own utility function shape and the utility decays exponentially. Also, the study in this chapter models special-purpose machine and task types, has different arrival patterns for the different kinds of tasks, and experiments with dropping low utility-earning tasks in an oversubscribed system.

The concepts of utility functions have been used in real-time systems for scheduling tasks [5, 6]. The problem of scheduling non-preemptive and mutually independent tasks on a single processor has been examined [5]. In that study, each task has a time value function that gives the task's contribution at its completion time. The goal is to order the execution of the tasks on the single processor to maximize the cumulative contribution from the tasks. Analytical methods have been used to create performance features and optimize them [6]. In that study, all jobs have the same shape for their utility functions, as opposed to this study where every task can have a different shape for its utility function. Although these papers address the maximization of total utility earned, the environment of a single processor versus the environment of a heterogeneous distributed system makes solution techniques significantly different for the two cases.

In [7], the users of a homogeneous high performance computing system can draw arbitrary shapes for utility functions for the jobs they submit. The users decide the level of accuracy in modeling the utility functions. The work in [7] uses a genetic algorithm to solve the problem of maximizing utility. The average execution time of the algorithm is 8,900 seconds. In this study, scheduling decisions are made at much smaller intervals (after a minute in the case of the batch-mode heuristics). Furthermore, a heterogeneous computing system is assumed, as opposed to the homogeneous computing system that they model.

2.6. Simulation Setup

2.6.1. Overview. In this study, a heterogeneous computing environment where a workload of tasks arrive dynamically is simulated. To model the execution time characteristics of the workload, an Estimated Time to Compute (ETC) matrix (as described in Sec. 2.2.2) is used. To completely describe the workload, one has to determine each task's utility function parameters, task type, and arrival time. In this section, it is explained how these parameters are generated for the simulations *based on the expectations for future environments of DOE and DoD interest*.

Each experiment discussed in Sec. 2.7 has its results averaged over 50 simulation trials. Each trial has a new workload of tasks (with different utility functions, task types, and arrival times). Each trial models a different compute environment by using different values for the entries of the ETC matrix. The method of generating these values for each of the trials will now be described.

2.6.2. Generating Utility Functions. For each task in the workload, the three parameters to describe its utility function (i.e., priority, urgency, and utility class) need to be assigned. As mentioned in Sec. 2.2.1.2, there are four possibilities for each of these

parameters. Four utility classes in this study are modeled because they are representative of the expected workload at ESSC. In the simulations, a task's utility class is chosen uniformly at random among the four classes modeled. Fig. 2.4 illustrates the utility functions obtained by using the four utility classes that are used in this study for a fixed priority level and a fixed urgency level. The length of the first interval during which the utility value does not decay is represented by "F" in the figure. It is dependent on the urgency level of the task as well as the average execution time of the task. App. B gives the method for its computation. App. C gives the values used to create the four utility classes.

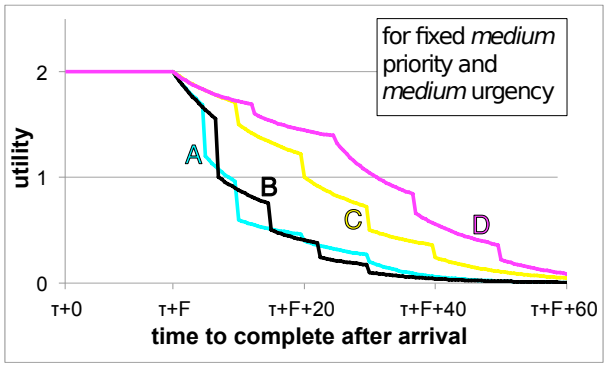


FIGURE 2.4. The utility functions of the four utility classes (A, B, C, and D) used in this study shown at fixed priority and urgency levels showing the decay in utility for a task after its arrival time τ. The duration of its first interval during which the utility value remains constant is represented by F on the x-axis.

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

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

2.6.3. Generating Estimated Time to Compute (ETC) Matrices. In the simulation environment, tasks that have similar execution time characteristics are grouped together into task types, and machines that have similar performance capabilities are grouped together into machine types. 100 task types and 13 machine types are modeled. In the simulations, the procedure by which tasks are assigned to task types is described in Sec. 2.6.4. An environment consisting of 100 machines is modeled, where each machine belongs to one of 13 machine types. Among these 13 machine types, 4 are special-purpose machine types while the remaining are general-purpose machine types. The special-purpose machine types are modeled as having the capability of executing certain task types (which are special to them) approximately ten times faster than on the general-purpose machine types. These special-purpose machine types, however, lack the ability to execute the other task types. In this environment, three to five task types were special on each special-purpose machine type.

Techniques from the Coefficient of Variation (COV) method [35] were used to generate the entries of the ETC matrix. The mean value of execution time on the general-purpose and the special-purpose machine types is set to ten minutes and one minute, respectively. Complete details about the parameters for generating ETC matrices are described in App. E. The appendix also discusses how the 100 machines are distributed among the 13 machine types.

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

2.6.4. Generating the Arrival Pattern of Tasks. To generate the arrival times of the tasks in the simulation, different arrival patterns for the special-purpose and the general-purpose task types are used. The goal of the arrival pattern generation is to closely

model expected workloads of DOE and DoD interest. The simulation models the arrival and mapping of tasks for a 24 hour period. Real-world oversubscribed systems rarely start with empty queues, so the arrival and mapping of tasks for 26 hours is simulated, and the first two hours of data is excluded from result calculations. The initial two hours serve to bring the system up to steady-state. All of the results (utility earned, average heuristic execution time, number of dropped tasks, etc.) are calculated for the duration of 24 hours (i.e., from the end of the 2^{nd} to the end of the 26^{th} hour). Based on the estimated level of oversubscription for the DOE/DoD environments of future interest, 33,000 task arrivals were simulated during a 24 hour period. To examine the impact on performance if the system were extremely oversubscribed, experiments were performed with 50,000 tasks arriving in a day.

Before the arrival patterns for the special-purpose and the general-purpose tasks types are generated, a mean arrival rate of tasks for every task type (irrespective of special-purpose or general-purpose) must be found. The estimated number of tasks of each task type that will arrive during the day is found by sampling from a Gaussian distribution. The mean for this distribution is the ratio of the desired number of tasks to arrive (33,000 or 50,000) to the number of task types in the system. The variance is set to $1/10^{th}$ of the mean. The mean arrival rate of a task type is obtained by dividing the estimated number of tasks of this task type that are to arrive during the period by 24 hours. The mean arrival rate of each task type is used to generate arrival rate patterns (that have different arrival rates during the 24 hours), based on whether it is a special-purpose or a general-purpose task type. For the general-purpose task types, a sinusoidal pattern is used for the arrival rate. For the special-purpose task types, a bursty arrival rate pattern is used. App. F discusses how the arrival pattern for a general-purpose or special-purpose task type is created, and how this

arrival rate pattern of a task type is used to obtain the actual number and arrival times of the tasks belonging to that task type.

2.7. Simulation Results and Analysis

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

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

The best value of K for the K-Best Types heuristic was empirically found to be K=1 machine type in this environment. At K=1, the K-Best Types performs the same mapping

decisions as the MET-Max Util heuristic. Therefore the results from this heuristic are not shown.

2.7.2. Preliminary Results. Fig. 2.5 shows the performance of the different heuristics in terms of the percentage of maximum utility earned with the two levels of oversubscription. Irrespective of the oversubscription level, one can observe that the naive immediate-mode heuristics always perform poorly compared to the smart immediate-mode heuristics. This is because the naive heuristics do not consider ETC information, machine ready-times, and the utility earned by a task on the various machines. The batch-mode heuristics always perform significantly better than the smart immediate-mode heuristics. This is because the batch-mode heuristics not only consider machine ready-times, but also have the ability to schedule a set of tasks and re-map tasks that are in the virtual queues. Most of the batch-mode heuristics are able to use this to their advantage and move any high utility-earning task that may have just arrived to the front of the virtual queues in the next mapping event. With the immediate-mode heuristics, the newly-arrived high utility-earning tasks would be queued behind other tasks, and by the time they get an opportunity to execute, their utility may have decayed significantly. With the 33,000 tasks per day case, on average, the batch-mode heuristics gave an improvement of approximately 250% compared to the smart immediate-mode heuristics.

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

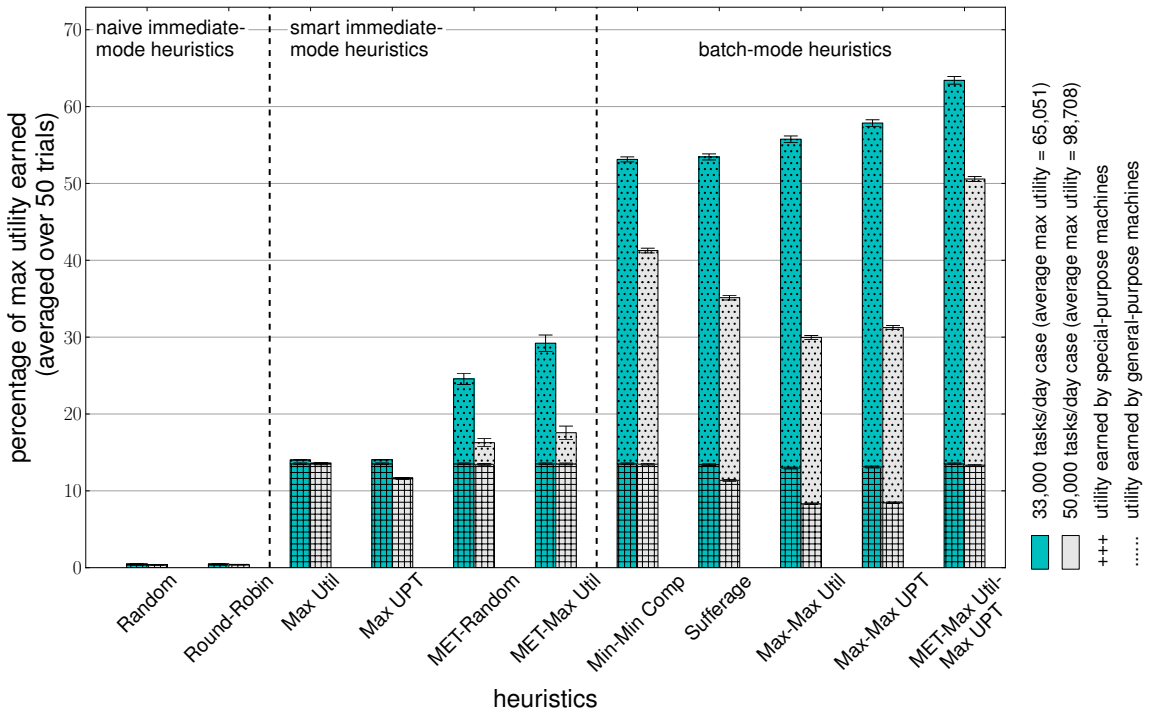


FIGURE 2.5. Percentage of maximum utility earned by all the heuristics under two levels of oversubscription: 33,000 tasks arriving within a day, and 50,000 tasks arriving within a day. No tasks were dropped in these cases. The utility earned value (as opposed to the percentage of maximum utility earned) by a heuristic in the 50,000 tasks per day case will typically be higher than that in the 33,000 tasks per day case.

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

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

problem by assigning tasks to machines where they execute the fastest. This allows these heuristics to earn utility from the general-purpose machines as well.

The performance of many batch-mode heuristics is severely affected by the increase in the oversubscription level. The higher oversubscription results in more tasks being present in the batch during the mapping events. With an increase in the size of the batch, the batch-mode heuristics take considerably longer to perform each mapping event. This leads to triggering fewer mapping events (because a new mapping event cannot begin until the previous one completes). Fig. 2.7 shows the total number of mapping events for the batch-mode heuristics under the two levels of oversubscription. The total number of mapping events are partitioned into two sections: those triggered at the time interval of one minute and those initiated when the execution of the previous mapping event took longer than one minute. One can observe that the batch-mode heuristics (other than Min-Min Comp in 33,000 tasks per day case) have fewer mapping events being triggered than the expected amount (namely, 1,440 if they were all triggered after one minute). With fewer mapping events, it takes longer for the high utility-earning tasks to be moved up to the front of the virtual queues and the delay may cause their utility values to decay significantly. Min-Min Comp executes faster than the other batch-mode heuristics because it does not perform any explicit utility calculations. MET-Max Util-Max UPT also executes relatively quickly because it performs utility calculations only for a subset of the machines. Max-Max Util and Max-Max UPT earn very low utility in the 50,000 tasks per day case because they have only 200 mapping events being triggered during the day. In contrast to the batch-mode heuristics, the immediate-mode heuristics execute quickly, and as a result, even in the case where 50,000 tasks arrive during the day, they have approximately 50,000 mapping events with only 0.5% of those on average (250

out of 50,000) being initiated as a result of the heuristic execution of the previous mapping event taking longer than the arrival time of the next task.

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

2.7.3. Results with Dropping Tasks. As mentioned in Sec. 2.4.4, techniques to drop tasks that earn utility values less than a dropping threshold in the immediate-mode and batch-mode heuristics were implemented. Experiments for six levels for the dropping threshold were performed: 0 (which is equivalent to no dropping), 0.05, 0.5, 1.5, 3, and 5. These are chosen based on the system model, including the values of maximum utility for the various priority levels, i.e., 8, 4, 2, and 1. Simulations with all the heuristics using the six dropping thresholds for the two cases of oversubscription were run. In Fig. 2.6, the results for the 50,000 tasks per day case are shown. The results of the 33,000 oversubscription level show similar trends, and are discussed in App. G. The heuristics significantly benefit from the dropping operation. For almost all heuristics, the utility earned increases as the dropping threshold increases from 0 to 1.5. With a dropping threshold of 1.5, all the *low* priority tasks are dropped because their starting utility is 1. This may be undesirable in general, but for this oversubscribed system this results in the best performance. The average computation

capability of this environment is such that approximately 26,000 tasks can execute in the 24-hour period (based on the average execution time of each task on each machine). The dropping operation lets us pick the best 26,000 tasks to execute to maximize the total utility that can be earned. Based on a different system model and administrative policies one may set the specific levels of dropping thresholds differently.

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

There are two main reasons why the batch-mode heuristics benefit from the dropping operation. The first is that the dropping operation helps them reduce the size of their batch during each mapping event by dropping tasks that would only be able to earn low utility. This makes the mapping events execute faster and results in more mapping events. For the batch-mode heuristics, in all cases where some level of dropping was implemented, all of the 1440 mapping events were triggered. With the increase in the number of mapping events, the batch-mode heuristics are able to service high utility-earning tasks faster. This causes

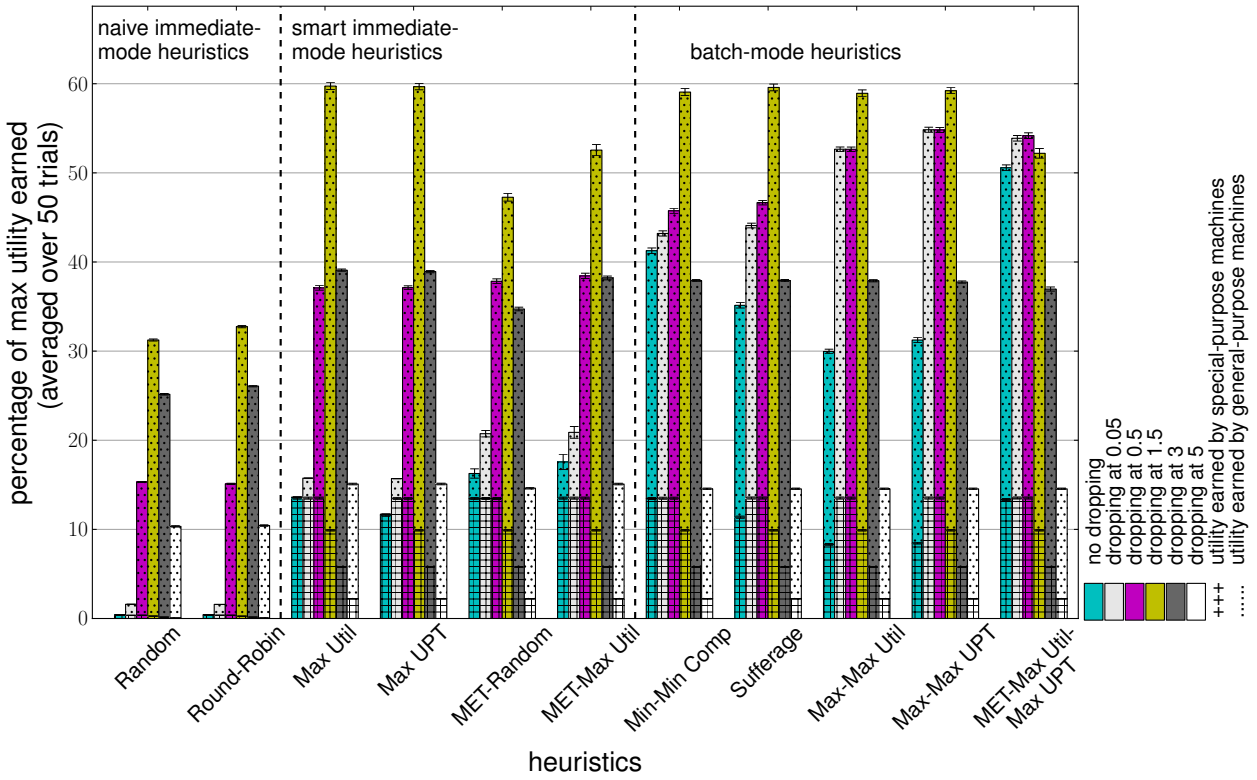


FIGURE 2.6. Percentage of maximum utility earned earned by all the heuristics for the different dropping thresholds with the oversubscription level of 50,000 tasks arriving during the day. The average maximum utility bound for this oversubscription level is 98,708.

the improvement in performance in the dropping at 0.05 case compared to the no dropping case. The second reason the batch-mode heuristics benefit from the dropping operation is the prevention of low utility-earning tasks from blocking the pending and the executing slots of the machines. When tasks are arriving, there may be periods when most of the arriving tasks are neither *critical* nor *high* priority tasks. During this time period, other lower priority tasks get the opportunity to fill into the pending slots of the machines. If there is a burst of *critical* or *high* priority tasks after this period, these higher-priority tasks will have to wait in queue behind the lower priority task in the pending slot, because the pending slot tasks cannot be re-mapped. By dropping the lower priority tasks, the pending (and hence the executing) slots are not occupied and when the high utility-earning tasks arrive they get to

quickly start execution and provide higher utility to the system. This causes the performance improvement for batch-mode heuristics with further dropping beyond 0.05. Similar to the immediate-mode heuristics, dropping thresholds greater than 1.5 drop too many tasks.

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

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

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

Although the smart immediate-mode heuristics can earn utility comparable to the batch-mode heuristics, their performance is very sensitive to the value of the dropping threshold. For the immediate-mode heuristics, the dropping threshold parameter needs to be tuned based on the starting utility values for the different priority levels, arrival pattern of the tasks, degree of oversubscription of the environment, etc., because the immediate-mode heuristics rely on the dropping threshold to empty the machine queues. In contrast, the

mechanism by which the dropping operation helps batch-mode heuristics such as Max-Max Util, Max-Max UPT, and MET-Max Util-Max UPT is different, i.e., it increases the number of mapping events. The performance of these batch-mode heuristics is less sensitive to the value of the dropping threshold.

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

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

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

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

Fig. 2.8 shows the performance of the Max-Max UPT heuristic with the different cases of triggering. The other batch-mode heuristics show similar trends as the Max-Max UPT heuristic. In all five triggering cases mentioned above and for all of the batch-mode heuristics,

the average execution time of a mapping event with a dropping threshold of 0.5 is under 350 milliseconds.

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

2.8. Conclusions

In this chapter, a flexible metric that uses utility functions to compare the performance of resource allocation heuristics in an oversubscribed heterogeneous computing environment where tasks arrive dynamically throughout a 24 hour period was designed. This type of environment is modeled based on the needs of the ESSC at ORNL. Seven immediate-mode heuristics and five batch-mode heuristics are designed and their performance is analyzed in a simulated environment based on the total utility they could earn during a one day time period. It was observed that without the ability to drop tasks, the naive immediate-mode heuristics perform poorly compared to the smart immediate-mode heuristics, which

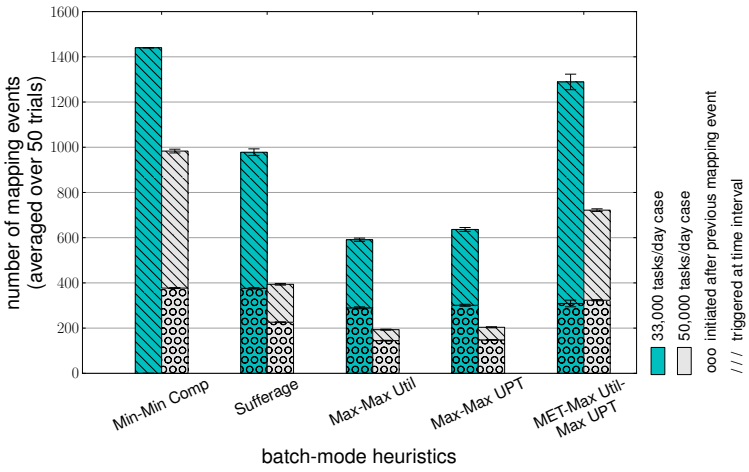


FIGURE 2.7. The number of mapping events initiated either because the one minute time interval has passed since the last mapping event or because the previous mapping event finished execution after one minute are shown for five batch-mode heuristics with the two levels of oversubscription: 33,000 tasks arriving within a day, and 50,000 tasks arriving within a day. No tasks were dropped in these cases.

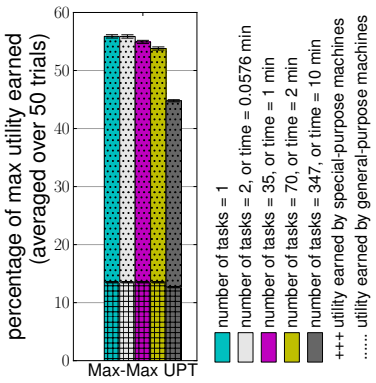


FIGURE 2.8. Percentage of maximum utility earned by the Max-Max UPT heuristic for the different cases of triggering batch-mode mapping events. The other batch-mode heuristics show similar trends.

in turn perform poorly compared to the batch-mode heuristics. Among the batch-mode heuristics, Max-Max UPT and MET-Max Util-Max UPT perform the best. This is because these batch-mode heuristics consider the minimization of the execution time of the task in addition to maximizing utility. This is helpful in an oversubscribed highly heterogeneous environment. Dropping low utility-earning tasks significantly helps improve performance of the immediate-mode heuristics because it allows other relatively high-utility earning tasks to execute sooner and thus earn more utility. Dropping tasks also improves the batch-mode heuristics in two ways, (a) by preventing large batch sizes which results in more mapping

events being triggered due to faster heuristic execution times, and (b) by preventing lower-priority tasks from entering into the pending slot so that higher priority tasks that arrive subsequently can execute sooner. Immediate-mode heuristics are much more sensitive to the value of the dropping threshold and rely on its tuning to avoid low utility earning tasks from entering machine queues. Permuting the initial tasks at the head of the virtual queues does not affect the performance significantly in this environment. Experiments with different triggers for the batch-mode mapping events were also performed. It was observed that (in this environment) triggering every time a new task arrives is not providing significant benefit in the total utility earned compared to mapping after every minute. Possible directions for future research are presented in Chapter 9.

CHAPTER 3

Deterministic Dynamic Utility Maximization with an Energy Constraint[1]

3.1. Introduction

During the past decade, large-scale computing systems have become increasingly powerful. As a result, there is a growing concern with the amount of energy needed to operate these systems [37, 38]. An August 2013 report by Digital Power Group estimates the global Information-Communications-Technologies ecosystem's use of electricity was approaching 10% of the world electricity generation [39]. As another energy comparison, it was using about 50% more energy than global aviation [39]. In 2007, global data center power requirements were 12 GW and in the four years to 2011, it doubled to 24 GW. Then, in 2012 alone it grew by 63% to 38 GW according to the 2012 DatacenterDynamics census [40]. Some data centers are now unable to increase their computing performance due to physical limitations on the availability of energy. For example, in 2010, Morgan Stanley, a global financial services firm based in New York, was physically unable to draw the energy needed to run a data center in Manhattan [41]. Many high performance computing (HPC) systems are now being forced to execute workloads with severe constraints on the amount of energy available to be consumed.

[1]This work was done jointly with Ph.D. student Bhavesh Khemka. The full list of co-authors is at [36]. This work was supported by Oak Ridge National Laboratory and their Extreme Scale Systems Center under subcontract numbers 4000094858 and 4000108022, and by a National Science Foundation Graduate Research Fellowship, and by NSF Grant CCF-1302693. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research also used resources of the Extreme Scale Systems Center at Oak Ridge National Laboratory as well as the CSU ISTeC Cray System supported by NSF Grant CNS-0923386.

The need for ever increasing levels of performance among HPC systems combined with higher energy consumption and costs are making it increasingly important for system administrators to adopt energy-efficient workload execution policies. In an energy-constrained environment, it is desirable for such policies to maximize the performance of the system. This research investigates the design of energy-aware scheduling techniques with the goal of maximizing the performance of a workload executing on an energy-constrained HPC system.

Specifically, a compute facility and workload of interest to the Extreme Scale Systems Center (ESSC) at Oak Ridge National Laboratory (ORNL) is modeled. The ESSC is a joint venture between the United States Department of Defense (DoD) and Department of Energy (DOE) to perform research and deliver tools, software, and technologies that can be integrated, deployed, and used in both DoD and DOE environments. The goal is to design resource management techniques that maximize the performance of their computing systems while obeying a specified energy constraint. Each task has a monotonically-decreasing utility function associated with it that represents the task's utility (or value) based on the task's completion time. The system performance is measured in terms of cumulative utility earned, which is the sum of utility earned by all completed tasks [2]. The example computing environment modeled, is based on the expectations of future DoD and DOE environments, incorporates heterogeneous resources that utilize a mix of different machines to execute workloads with diverse computational requirements. Several heterogeneous environments are created and studied that are very similar to this example environment but have different heterogeneity characteristics, as quantified by a Task-Machine Affinity (TMA) measure [42]. TMA captures the degree to which some tasks are better suited on some unique machines. An environment where all tasks have the same ranking of machines in terms of execution time has zero TMA. In an environment with high TMA, different tasks will most likely have a

unique ranking of machines in terms of execution time. It is important to analyze the impact on performance if the TMA of the environment is changed. The performance of low and high TMA environments is modeled, analyzed, and compared to the example environment based on interests of the ESSC. This analysis also can help guide the selection of machines to use in a computing system (based on the expected workload of tasks) to maximize the performance obtainable from the system.

In a heterogeneous environment, tasks typically have different execution time and energy consumption characteristics when executed on different machines. Machines are modeled to have three different performance states (P-states) in which tasks can execute. By employing different resource allocation strategies, it is possible to manipulate the performance and energy consumption of the system to align with the goals set by the system administrator. To keep simulations tractable, only the energy consumed by the system on a daily basis with the goal of meeting the annual energy constraint is considered. Four novel energy-aware resource allocation policies that have the goal of maximizing the utility earned while obeying an energy constraint over the span of a day were developed. These policies are compared with three techniques from the literature designed to maximize utility [2] and it is shown that for energy-constrained environments, heuristics that manage their energy usage throughout the day outperform heuristics that only try to maximize utility. We enhance the resource allocation policies by designing an energy filter (based on the idea presented in [10]) for this environment. The goal of the filtering technique is to remove high energy consuming allocation choices that use more energy than an estimated fair-share. This step improves the distribution of the allotted energy across the whole day. An in-depth analysis to demonstrate the benefits of the energy filter is performed. The performance of all the

heuristics in the low and high TMA environments is studied and extensive parameter tuning tests are performed.

In summary, the following contributions are made: (a) the design of four new resource management techniques that maximize the utility earned, given an energy constraint for an oversubscribed heterogeneous computing system, (b) the design of a custom energy filtering mechanism that is adaptive to the remaining energy, enforces "fairness" in energy consumed by tasks, and distributes the energy budgeted for the day throughout the day, (c) a method to generate new heterogeneous environments that have low and high TMA compared to the environment based on interests of the ESSC without changing any of its other heterogeneity characteristics, (d) show how heuristics that only maximize utility can become energy-aware by adapting three previous techniques to use an energy filter, (e) a sensitivity analysis for all the heuristics to the parameter that controls the level of energy-awareness and/or level of energy filtering, (f) an analysis of the performance of all the heuristics in the low and high TMA environments, and (g) a recommendation on how to select the best level of filtering or the best balance of energy-awareness versus utility maximization for heuristics based on a detailed analysis of the performance of the heuristics.

The remainder of this chapter is organized as follows. The next section formally describes the problem and the system model. Section 3.3 describes the resource management techniques. An overview of related work is provided in Section 3.4. The simulation and experimental setup are detailed in Section 3.5. In Section 3.6, the results of the experiments are discussed and analyzed. The conclusion is presented in Section 3.7.

3.2. Problem Description

3.2.1. System Model. In this study, we assume a workload where tasks arrive dynamically throughout the day and the resource manager maps the tasks to machines for execution. We model our workload and computing system based on the interests of the ESSC. Each task in the system has an associated utility function (as described in [2]). Utility functions are monotonically-decreasing functions that represent the task's utility (or value) of completing the task at different times. We assume the utility functions are given and can be customized by users or system administrators for any task.

Our computing system environment consists of a suite of heterogeneous machines, where each machine belongs to a specific machine type (rather than a single large monolithic system, such as Titan [43]). Machines belonging to different machine types may differ in their microarchitectures, memory modules, and/or other system components. We model the machines to contain CPUs with dynamic voltage and frequency scaling (DVFS) enabled to utilize three different performance states (P-states) that offer a trade-off between execution time and power consumption. We group tasks with similar execution characteristics into task types. Tasks belonging to different task types may differ in characteristics such as computational intensity, memory intensity, I/O intensity, and memory access pattern. The type of a task is not related to the utility function of the task. Because the system is heterogeneous, machine type A may be faster (or more energy-efficient) than machine type B for certain task types but slower (or less energy-efficient) for others.

We assume that for a task of type i on a machine of type j running in P-state k, we are given the Estimated Time to Compute ($ETC(i,j,k)$) and the Average Power Consumption ($APC(i,j,k)$). It is common in the resource management literature to assume the availability of this information based on historical data or experiments [11–15, 44, 16]. The

APC incorporates both the static power (not affected by the P-state of the task) and the dynamic power (different for different P-states). We can compute the Estimated Energy Consumption ($EEC(i,j,k)$) by taking the product of execution time and average power consumption, i.e., $EEC(i,j,k) = ETC(i,j,k) \times APC(i,j,k)$. We model general-purpose machine types and special-purpose machine types [45]. The special-purpose machine types execute certain special-purpose task types much faster than the general-purpose machine types, although they may be incapable of executing the other task types. Due to the sensitive nature of DoD operations, for the ESSC environment, historical data is not publicly available. Therefore, for the simulation study conducted in this chapter, we synthetically create our ETC and APC matrices based on the recommendations provided by ESSC.

Tasks are assumed to be independent (they do not require inter-task communication) and can execute concurrently (each on a single machine, possibly with parallel threads). This is typical of many environments such as [46]. We do not allow the preemption of tasks, i.e, once a task starts execution, it must execute until completion.

We model three degrees of heterogeneity by varying the TMA of the system [42]. TMA uses singular value decomposition (SVD) for its computation and captures the degree to which certain tasks execute faster on certain unique machines. Section 3.5.3.2 describes how TMA is computed. Task Difficulty Homogeneity (TDH) and Machine Performance Homogeneity (MPH) are given as orthogonal metrics for quantifying the heterogeneity of the system [42]. TDH and MPH capture the homogeneity in the aggregate performance of tasks and machines, respectively. We study the performance of all the heuristics in an example environment (based on the interests of the ESSC), a modified environment with low TMA, and a modified environment with high TMA. The low and high TMA environments differ only

in their TMA compared to the example environment. All three environments have similar values of TDH and MPH.

In an environment with extremely low TMA, all tasks will have the same sorted ordering of machines in terms of execution time. The actual execution time values of different tasks on the machines can be different, but all the tasks will rank the machines in the same order. In contrast, in an extremely high TMA environment, each task will have a unique ordering of machines if the machines were to be sorted in terms of execution time for that task. Environments that have different TMA but similar MPH and TDH do not have more powerful or less powerful machines or more difficult or less difficult tasks in general, but instead they differ in the level of uniqueness of the affinity of different tasks to the machines. An environment with higher TMA does not have more powerful machines, but instead has machines that are suited to perform well for different tasks. On the contrary, in an environment with lower TMA, it is easy to rank machines in terms of performance (irrespective of the tasks). It is desirable to analyze the performance of these different types of systems. Given the expected workload for an environment, such analyses can help guide the selection of resource management heuristics and associated tuning parameters in use. In Section 3.5.3.2, we describe our technique to create the relatively low and high TMA environments with negligible difference in the values of MPH and TDH of the system. Table 3.1 mentions the different acronyms used in this study and their meanings.

3.2.2. Problem Statement. The amount of useful work accomplished by a compute system is captured by the total utility it earns from task completions. Recall that we consider a workload model where tasks arrive dynamically. The resource manager does not know which task will arrive next, the utility function of the task, nor its task type. Therefore,

TABLE 3.1. Acronyms Used and Their Meanings

acronyms	**meaning**
ORNL	Oak Ridge National Laboratory
DoD	United States Department of Defense
DOE	United States Department of Energy
ESSC	Extreme Scale Systems Center at ORNL
TMA	Task-Machine Affinity
TDH	Task Difficulty Homogeneity
MPH	Machine Performance Homogeneity
DVFS	Dynamic Voltage and Frequency Scaling
P-states	Performance states of a processor
ETC	Estimated Time to Compute
APC	Average Power Consumed
EEC	Estimated Energy Consumption
ECS	Estimated Computation Speed
U-E wf	Utility-Energy Weighting Factor

the goal of the resource manager is to maximize the total utility that can be earned from completing tasks while satisfying an annual energy constraint.

Many computing centers (including the ESSC environment that we consider) have budget constraints on annual expenditure towards energy consumption of the compute system. This annual energy-cost budget can be translated into an annual energy constraint. To simplify the problem, we divide the annual energy constraint into daily energy constraints and ensure that a given day's energy constraint is met. If models are provided that give the variation in energy cost with the time of the year and give the expected workload arriving into the system during the different times of the year, one could use such information to calculate an appropriately scaled value for a given day's energy constraint ($\underline{energy\ constraint_{day}}$). If such models are not available, one can approximate $energy\ constraint_{day}$ by dividing the total energy remaining for the year and the number of days remaining in the year. This reduces the problem to that of maximizing the total utility earned per day while obeying $energy\ constraint_{day}$. This artificial energy constraint allows the system to control its energy consumption on a per-day basis to be able to meet the actual energy constraint, which is

for the year. We use the duration of a day to keep the simulation time tractable. Instead of one day, we could base our constraint on any interval of time (e.g., two hours, six months, a year). If a task starts execution on one day and completes execution on the next, the utility earned and the energy consumed for each day is prorated based on the proportion of the task's execution time in each day. This is done so that each day gets the utility for the task executions that consumed its energy to permit a fair comparison of different heuristic approaches. Systems such as the ESSC are designed so that constraints on power (i.e., energy per unit time) are not a concern.

3.3. Resource Management

3.3.1. Overview. It is common to use heuristics for solving the task to machine resource allocation problem as it has been shown, in general, to be NP-complete [17]. A mapping event occurs any time a scheduling decision has to be made. We use batch-mode heuristics that trigger mapping events after fixed time intervals as they performed best in our previous work [2]. To account for oversubscription (i.e., more tasks arrive than can possibly be executed while they are still valuable), we use a technique that drops tasks with low potential utility at the current time. We also design an energy filter that helps guarantee the energy constraint by avoiding allocating tasks that use more than their "fair-share" of energy. Our simulation results show the benefit of this technique.

Mapping events for our batch-mode heuristics are triggered every minute. If the execution of the previous mapping event takes longer than a minute, then the next mapping event is triggered after the previous one completes execution. The task that is next-in-line for execution on a machine is referred to as the pending task. All other tasks that are queued for the machines are said to be in the virtual queues of the scheduler. Figure 3.1 shows

a small example system with four machines, the executing tasks, the tasks in the pending slots, and the virtual queues of the scheduler. At a mapping event, the batch-mode heuristics make scheduling decisions for a set of tasks comprising the tasks that have arrived since the last mapping event and the tasks that are currently in the virtual queues. This set of tasks is called the mappable tasks set. The batch-mode heuristics are not allowed to remap the pending tasks so that the machines do not idle if the currently executing tasks complete while the heuristic is executing. In this study, we adapt three batch-mode heuristics (from our previous work [2]) to the new environment, design four new energy-aware batch-mode heuristics, and analyze and compare their performances.

The scheduling decisions made for a task may depend on the energy that currently remains in the system for the day (among other factors). The value of the per day energy constraint can change each day based on the energy that has been consumed thus far in the year. Therefore, we want to avoid making scheduling decisions for a task that starts its execution on the next day. Therefore, the batch-mode heuristics are not allowed to map a task to a machine where it will start execution on the next day. In addition, the batch-mode heuristics are not allowed to schedule a task that will violate the day's energy constraint.

If no machine can start the execution of the task within the current day or if the task will violate the current day's energy constraint, then the task's consideration is postponed to the next day. At the start of the next day, all postponed tasks are added to the mappable tasks set and the heuristics make mapping decisions for these tasks as well.

3.3.2. Batch-mode Heuristics. We present four new heuristics that try to maximize the utility earned while being energy-aware. The Max-Max Utility-Per-Energy consumption (**Max-Max UPE**) heuristic is a two-stage heuristic based on the concept of the two-stage Min-Min heuristic that has been widely used in the task scheduling literature [19, 20, 26,

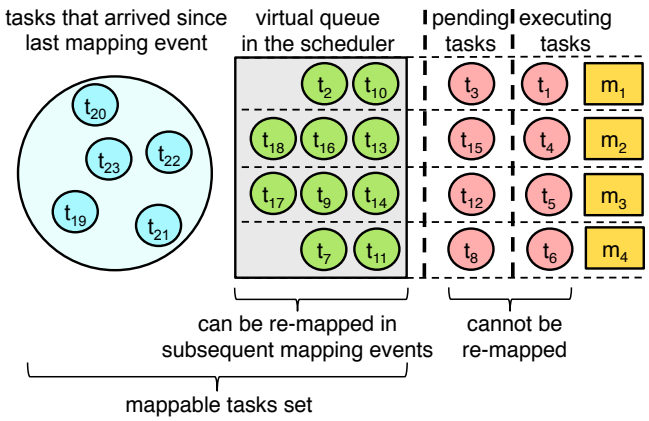


FIGURE 3.1. An example system of four machines showing tasks that are currently executing, waiting in pending slots, waiting in the virtual queue, and have arrived since the last mapping event (and are currently unmapped).

24, 25, 47, 27, 8, 9, 29]. In the first stage, the heuristic finds for each task independently in the mappable tasks set the machine and P-state that maximizes "utility earned / energy consumption." If none of the machine-P-state choices for this task satisfy the day's energy constraint nor start the execution of the task within the current day, then, the task is postponed to the next day and removed from the mappable tasks set. In the second stage, the heuristic picks the task-machine-P-state choice from the first stage that provides the overall highest "utility earned / energy consumption." The heuristic assigns the task to that machine, removes that task from the set of mappable tasks, updates the machine's ready time, and repeats this process iteratively until all tasks are either mapped or postponed.

The Weighted Utility (**Weighted Util**) heuristic is designed to explicitly control the extent to which allocation decisions should be biased towards maximization of utility versus minimization of energy consumption. It does this by using a utility-energy weighting factor (*U-E weighting factor* or *U-E wf*) that controls the relative significance of normalized utility and normalized energy in the heuristic's objective function. To normalize the utility value across different allocation choices, we divide by the maximum utility any task in the system could have (*max util*). For normalizing energy consumption, we determine the highest EEC

value from all possible task, machine, and P-state combinations (*max energy consumption*). The Weighted Util heuristic is also a two-stage heuristic. In the first stage, for each mappable task, it finds the machine-P-state pair that has the highest value for the weighted expression:

$$(1 - \textit{U-E } wf) \times \frac{\textit{utility earned}}{\textit{max util}} - \textit{U-E } wf \times \frac{\textit{energy consumed}}{\textit{max energy consumption}}. \tag{3}$$

As done in Max-Max UPE, if no allocation choices for this task satisfy the day's energy constraint or start the execution of the task within the current day, then, the task is removed from the mappable tasks set and is postponed to be considered the next day. In the second stage, it picks from the pairs of the first stage the one that has the highest value for the above expression, makes that assignment, removes that task from the set of mappable tasks, updates that machine's ready time, and repeats the two stages iteratively until all tasks have either been mapped or postponed.

By normalizing the utility and the energy terms in the weighted expression, we ensure that each of those terms has a value between 0 and 1. This makes it convenient to compare utility and energy in a single expression as we do. The value of *U-E* wf can be varied between 0 and 1 to bias the effect of the normalized energy term to the value of the weighted expression.

The Weighted Utility-Per-Time (**Weighted UPT**) heuristic is similar to the Weighted Util heuristic but the normalized utility term in expression 3 is replaced by "normalized utility earned / normalized execution time." To normalize the execution time, we determine the minimum execution time in the ETC matrix from all task, machine, P-state choices

(*min execution time*). The weighted expression for this heuristic is:

$$(1 - \textit{U-E } wf) \times \frac{\textit{utility earned/execution time}}{\textit{max util/min execution time}} - \textit{U-E } wf \times \frac{\textit{energy consumed}}{\textit{max energy consumption}}. \tag{4}$$

The Weighted Utility-Per-Energy consumption (**Weighted UPE**) heuristic is similar to the Weighted Util heuristic but the normalized utility term in expression 3 is replaced by "normalized utility earned / normalized energy consumption." To normalize energy, we determine the minimum energy consumption value in the EEC matrix from all task, machine, P-state choices (*min energy consumption*). The weighted expression for this heuristic is:

$$(1 - \textit{U-E } wf) \times \frac{\textit{utility earned/energy consumed}}{\textit{max util/min energy consumption}} - \textit{U-E } wf \times \frac{\textit{energy consumed}}{\textit{max energy consumption}}. \tag{5}$$

For comparison, we analyze the following three utility maximization heuristics to examine how heuristics that do not consider energy perform in an energy-constrained environment. These heuristics assign tasks while there still remains energy in the day.

The Min-Min Completion time (**Min-Min Comp**) heuristic is a fast heuristic adapted from [2] and is a two-stage heuristic like the Max-Max Utility-Per-Energy heuristic. In the first stage, this heuristic finds for each task the machine and P-state choice that completes execution of the task the soonest. This also will be the machine-P-state choice that earns the highest utility for this task (because we use monotonically-decreasing utility functions). In the second stage, the heuristic picks the task-machine-P-state choice from the first stage that provides the earliest completion time. This batch-mode heuristic is computationally efficient because it does not explicitly perform any utility calculations

The Max-Max Utility (**Max-Max Util**) heuristic introduced in [2] is also a two-stage heuristic like the Min-Min Comp heuristic. The difference is that in each stage Max-Max Util maximizes utility, as opposed to minimizing completion time. In the first stage, this heuristic finds task-machine-P-state choices that are identical to those found in the first stage of the Min-Min Comp heuristic. In the second stage, the decisions made by Max-Max Util may differ from those of Min-Min Comp. This is because picking the maximum utility choice among the different task-machine-P-state pairs depends both on the completion time and the task's specific utility function.

The Max-Max Utility-Per-Time (**Max-Max UPT**) heuristic introduced in [2] is similar to the Max-Max Util heuristic, but in each stage it maximizes "utility earned / execution time," as opposed to maximizing utility. This heuristic selects assignments that earn the most utility per unit time, which can be beneficial in an oversubscribed system.

We collectively refer to the Weighted Util, Weighted UPT, and Weighted UPE heuristics as the weighted heuristics. We also collectively refer to the Min-Min Comp, Max-Max Util, Max-Max UPT, and Max-Max UPE heuristics as the non-weighted heuristics.

The weighted heuristics can be viewed as more generalized versions of their non-weighted counterparts. For example, the Weighted Util heuristic can be viewed as a more generalized version of the Max-Max Util heuristic. If *U-E* $wf = 0$, then Weighted Util reduces to Max-Max Util. For higher values of *U-E* wf, the Weighted Util heuristic is more energy-aware and has the goal of simultaneously maximizing utility while minimizing energy.

3.3.3. Dropping Low Utility Earning Tasks. We use a technique to drop tasks with low potential utility at the current time (introduced in our previous work [2]). Dropping a task means that it will never be mapped to a machine. This operation allows the batch-mode heuristics to tolerate high oversubscription. Due to the oversubscribed environment,

if a resource allocation heuristic tried to have all tasks execute, most of the task completion times would be so long that the utility of most tasks would be very small. This would negatively impact users as well as the overall system performance. Given the performance measure is the total utility achieved by summing the utilities of the completed tasks, dropping tasks leads to higher system performance, as well as more users that are satisfied.

The dropping operation reduces the number of scheduling choices to consider and therefore at a mapping event the dropping operation is performed before the heuristic makes its scheduling decisions. When a mapping event is triggered, we determine the maximum possible utility that each mappable task could earn on any machine assuming it can start executing immediately after the pending task is finished. If this utility is less than a dropping threshold (determined empirically), we drop this task from the set of mappable tasks. If the utility earned is not less than the threshold, the task remains in the mappable tasks set and is included in the batch-mode heuristic allocation decisions.

Because of oversubscription in our environment, the number of tasks in the mappable tasks set increases quickly. This can cause the heuristic execution time to be long enough to delay the trigger of subsequent mapping events. This results in poor performance because it now takes longer for the heuristics to service any high utility earning task that may have arrived. By the time the next mapping event triggers, the utility from this task may decay substantially. By dropping tasks with low potential utility at the current time, we reduce the size of the mappable tasks set and enable the heuristics to complete their execution within the mapping interval time (a minute). This allows the heuristics to move any new high utility-earning task to the front of the virtual queue to complete its execution sooner.

If a batch-mode heuristic postpones a task to the next day, a check is performed to make sure that the maximum possible utility that the task could earn (at the start of the next

day) is greater than the dropping threshold. If it is not, the task is dropped instead of being postponed.

3.3.4. Energy Filtering. The goal of our new energy filter technique is to remove potential allocation choices (task-machine-P-state combinations) from a heuristic's consideration if the allocation choice consumes more energy than an estimated fair-share energy budget. We call this budget the *task budget*. The value of the *task budget* needs to adapt based on the energy remaining in the day and the time remaining in the day. Therefore, the value of the *task budget* is recomputed at the start of every mapping event. We do not recompute the value of the *task budget* within a mapping event (based on the allocations made by the heuristic in that mapping event) because we want the *task budget* to only account for execution information that is guaranteed to occur (i.e., executing and pending tasks).

$$\begin{aligned} energy\ remaining = {} & energy\ constraint_{day} \\ & - energy\ consumed \\ & - energy\ scheduled \end{aligned} \tag{6}$$

We denote *energy consumed* as the total energy that has been consumed by the system in the current day, and *energy scheduled* as the energy that will be consumed by tasks queued for execution. At the start of a mapping event, the virtual queued tasks are removed from the machine queues and inserted into the mappable tasks set. Therefore, *energy scheduled* will account for the energy that will be consumed by all tasks that are currently executing and the tasks that are in the pending slot. The total energy that can be scheduled by

heuristics (without violating the day's energy constraint) is denoted by *energy remaining*. It is computed using Equation 6.

To estimate the *task budget*, the filter also needs to compute the time remaining in the day within which the above energy can be consumed. The availability time of a machine is set to either the completion time of the last task to be queued for the machine or the current time, whichever is greater. At the start of the mapping event, the last task to be queued for a machine will be the pending task. The total time remaining for computations (summed across machines) in the day is denoted as the *aggregate time remaining*. We compute it by summing across machines the difference between the end time of the day and the availability time of the machine. Figure 3.2 shows its computation for an example small-scale system with three machines. As shown, even though machine m3 is not executing a task after time 16, the available compute time from that machine is obtained by taking the difference between end of the day and the current time.

The average of the execution time values of all task types, machine types, and P-states is represented as *average execution time*. The energy filtering technique needs to estimate the total number of tasks that can be executed in the remaining part of the day. It does this by taking the ratio of *aggregate time remaining* and *average execution time*. The energy filter has to use *average execution time* because the resource manager is unaware of the type of tasks that may arrive or which machine or P-state they will be assigned to for the rest of the day.

To adjust the value of the *task budget* around its estimate, we use a multiplier called *energy leniency*. Higher values of the *energy leniency* imply more leeway for high energy allocation choices to pass through the filter, whereas a low value for the *energy leniency*

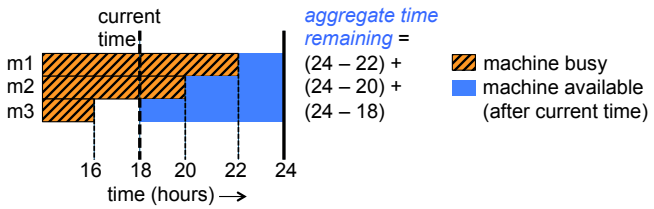


FIGURE 3.2. An example system of three machines showing the computation of *aggregate time remaining*. It represents the total computation time available from the current time till the end of the day.

would filter out many more choices. This value is determined empirically. The *task budget* is computed using Equation 7.

$$task\ budget = energy\ leniency \times \frac{energy\ remaining}{\left(\frac{aggregate\ time\ remaining}{average\ execution\ time}\right)} \tag{7}$$

This *task budget* is recomputed at the start of each mapping event and is an estimate of the amount of fair-share energy that we want an execution of a task to consume. At each mapping event, the heuristics consider only those task-machine-P-state allocation choices that consume less energy than the *task budget*.

3.4. Related Work

Heterogeneous task scheduling in an energy-constrained computing environment is examined in [8]. The authors model an environment where devices in an ad-hoc wireless network are limited by battery capacity and each task has a fixed priority. This differs significantly for our environment where we model a larger and more complex heterogeneous system with a utility performance metric based on the exact completion time of each task rather than a metric that aims to finish more higher priority tasks, as in their work. Additionally, in our

study, the energy available for use under the constraint is shared across all resources, while in [8] each resource has its own energy constraint (a battery).

In [2], the concept of utility functions to describe a task's time-varying importance is introduced. The authors deal with the problem of maximizing the total utility that can be earned from task completions. Energy is not considered at all in that paper. In this work, we are concerned with maximizing utility while obeying an energy constraint. For accomplishing this, we design four new energy-aware heuristics and an energy filtering technique that adapts to the remaining energy. We perform extensive analysis of all the heuristics along with parameter tuning tests. We also create low and high TMA environments and examine the performance of the heuristics in these environments.

An energy-constrained task scheduling problem in a wireless sensor network environment is studied in [48]. The authors analyze how the presence of an energy constraint affects the schedule length (i.e., makespan) when executing a set of dependent tasks. A wireless sensor network is significantly different from the environment we are modeling. In our model, each task contributes a certain amount of utility to the system. We are not concerned with minimizing a schedule length, as tasks continuously arrive through out the day.

In [49], a set of dynamically arriving tasks with individual deadlines are allocated to machines within a cluster environment with the goal of conserving energy. Specifically, the authors try to optimize the energy consumption while meeting the constraint of completing all tasks by their deadlines. Our environment tries to maximize the total utility earned while operating under an energy constraint. Additionally, [49] uses constant arrival patterns in an undersubcribed system, while our work focuses on highly oversubscribed environments where tasks arrive in varying sinusoidal or bursty patterns.

A dynamic resource allocation problem in a heterogeneous energy-constrained environment is studied in [10]. Tasks within this system contain individual deadlines, and the goal is to complete as many tasks by their individual deadlines as possible within an energy constraint. This is a different problem from our work as we are trying to maximize the utility earned (based on each task's completion time) and not the number of tasks that meet their hard deadlines. The authors of [10] use heuristics that map each task to a machine as soon as the task arrives and do not allow remapping, whereas in our environment we map groups of tasks at a time, allowing us to use more information when making allocation decisions and can also remap tasks in the virtual queue. The concept of an energy filter is used in [10], and we build on that for a more complex filter.

In [50], the authors formulate a bi-objective resource allocation problem to analyze the trade-offs between makespan and energy consumption. Their approaches use total makespan as a measure of system performance as opposed to individual task utility values as we do in our work. Additionally, they model static environments where the entire workload is a single bag-of-tasks, unlike our work that considers a system where tasks arrive dynamically. In our work, we consider maximizing the utility earned while meeting an energy constraint whereas [50] does not consider an energy constraint in its resource allocation decisions.

3.5. Simulation Setup

3.5.1. Overview. We simulate the arrival and mapping of tasks over a two day span with the first day used to bring the system up to steady-state operation. We collect our results (e.g., total utility earned, energy consumed) only for the second day to avoid the scenario where the machines start with empty queues. All the simulation experiments were run on the ISTeC Cray System at Colorado State University [46]. In this system, each node

has 24 cores. The nodes are scheduled to a user as a whole, and therefore, we average the results of our experiments across 48 simulation trials (instead of say 50). Each of the trials represents a new workload of tasks (with different utility functions, task types, and arrival times), and a different computing environment by using new values for the entries in the ETC and APC matrices (but without changing the number of machines). All of the parameters used in our simulations are set to closely match the expectations for future environments of interest to the ESSC.

3.5.2. Workload Generation. A utility function for each task in a workload is given and each task has a maximum utility value that starts at one of 8, 4, 2, or 1. These values are based on the plans of the ESSC, but for other environments, different values of maximum utility may be used. Furthermore, for our environment we have four choices of maximum utility but in other environments greater or fewer choices may be used. A method for generating utility functions can be found in [2].

For our simulation study, we generate the arrival patterns to closely match patterns of interest to ESSC [2]. In this environment, general-purpose tasks arrive in a sinusoidal pattern and special-purpose tasks follow a bursty arrival pattern.

3.5.3. Execution Time and Power Modeling.

3.5.3.1. *Example Environment.* This example environment is based on the expectation of some future DoD/DOE environments. In our simulation environment, approximately 50,000 tasks arrive during the duration of a day and each of them belongs to one of 100 task types. Out of the 100 task types, 83 were general-purpose and 17 were special-purpose task types. Each task type has approximately the same number of tasks in it. Furthermore, each task's utility function is generated using the method in [2]. The compute system that we model

has 13 machine types consisting of a total of 100 machines. Among these 13 machine types, four are special-purpose machine types while the remaining are general-purpose machine types. Each of the four special-purpose machine types has 2, 2, 3, and 3 machines on them, respectively, for a total of ten special-purpose machines. The remaining 90 machines are general-purpose and are split into the remaining nine machine types as follows: 5, 5, 5, 10, 10, 10, 10, 15, and 20. The machines of a special-purpose machine type run a subset of special-purpose task types approximately ten times faster on average than the general-purpose machines can run them (as discussed below). The special-purpose machines do not have the ability to run tasks of other task types. In our environment, three to five special-purpose task types are special for each special-purpose machine type.

We assume that all machines have three P-states in which they can operate. We use techniques from the Coefficient of Variation (COV) method [35] to generate the entries of the ETC and APC matrices in the highest power P-state. The mean value of execution time on the general-purpose and the special-purpose machine types is set to ten minutes and one minute, respectively. The mean value of the static power for the machines was set to 66 watts and the mean dynamic power was set to 133 watts. To generate the dynamic power values for the intermediate P-state and the lowest power P-state, we scale the dynamic power to 75% and 50%, respectively, of the highest power P-state. The execution time for these P-states are also generated by scaling the execution time of the highest power P-state. To determine the factor by which we will scale the execution time of the highest power P-state for the intermediate and lowest power P-states, we sample a gamma distribution with a mean value that is approximately $1/\sqrt{(\%\ scaled\ in\ power)}$. For example, the lowest power P-state's execution time will be scaled by a value sampled from a gamma distribution that has a mean approximately equal to $1/\sqrt{0.5}$. The execution time of any task is guaranteed

to be the shortest in the highest power P-state, but the most energy-efficient P-state can vary across tasks. These are done to model reality where the impact on execution time and energy consumption by switching P-states depends on the CPU-intensity/memory-intensity of the task, overhead power of the system, etc. We refer to this set of matrices as the example environment.

3.5.3.2. *Low and High TMA Environments.* We modify the ETC matrices at the highest power P-state of the example environment to create low and high TMA environments with minimal difference in the MPH and TDH of the environments. All three of these measures are functions of the Estimated Computation Speed (ECS) matrices. An ECS matrix is created by taking the reciprocal of each entry in the ETC matrix. Figure 3.3 shows two 3×3 ECS matrices that significantly differ in TMA but would have the same value for TDH. In the high TMA ECS matrix, each task has a unique ranking of the machines in terms of speed of execution.

To make the TMA measure orthogonal to the MPH and TDH measures, alternate row and column normalizations are performed on the ECS matrix so that the matrix has equal row sums and equal column sums (called a standard matrix) before the TMA is computed for the ECS matrix [42]. As mentioned in [42], this iterative procedure is not guaranteed to converge to a standard matrix if the ECS matrix can be organized into a block matrix (after reordering rows and columns) with one block containing only 0s. Such a matrix is said to be decomposable [42]. In our environment, the entries in the ECS matrix for the special-purpose machines and the tasks that are not special on them contain 0s (because the special-purpose machines are unable to execute them), and therefore, our ECS matrices are decomposable. To overcome this problem, we remove the columns of the special-purpose machines from the matrices and compute the TMA of only the general-purpose-machines matrix that has the

very low TMA	machine1	machine 2	machine 3
task 1	25	10	5
task 2	25	10	5
task 3	25	10	5

high TMA	machine1	machine 2	machine 3
task 1	25	10	5
task 2	5	25	10
task 3	10	5	25

FIGURE 3.3. Two sample 3×3 ECS matrices that have equal Task Difficulty Homogeneity but very different values of Task Machine Affinity. In the matrix with the high TMA, each task has a unique ranking of machines in terms of execution speed, i.e., for task 1 the best to worst machines are: 1, 2, and 3, whereas for task 2 the ranking of machines would be: 2, 3, and 1. In contrast, in the very low TMA matrix, all tasks would have the same ranking of machines.

general-purpose machines. We then modify this part of the ECS matrix to have low and high TMA. For each of the low and high TMA matrices, we then obtain the values for the special-purpose tasks (on the one machine on which they are special) by taking the average of the entries of this task from the general-purpose-machines matrix and multiplying that average speed by 10. By doing this, we retain the characteristics of the special-purpose machines and tasks as desired by the ESSC at ORNL, but we are able to study the performance of the heuristics in environments with different TMA. The TMA of the general-purpose-machines ECS matrices do not capture the actual TMA of the whole environment. However, as we are only concerned with creating relatively low and high TMA matrices compared to the example environment, our computation of the TMA measure is valid for our purposes.

As mentioned in [42], after a standard matrix is obtained (by performing alternate row and column normalizations), the first step for computing the TMA of the matrix is to determine the SVD. An ECS matrix with $\underline{T}$ task types and $\underline{M}$ machine types has dimension $T \times M$. The SVD results in the factorization $U\Sigma V^T$. The U and V^T are orthogonal matrices representing the column and the row space, respectively. The matrix Σ is a diagonal matrix

consisting of $min(T, M)$ singular values ($\underline{\sigma_i}$) along the diagonal in a monotonically decreasing order, i.e., $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_{min(T,M)} \geq 0$. The singular values represent the degree of correlation (linear dependence) between the columns of the ECS matrix. The higher the first singular value (σ_1), the more correlated the columns of the matrix are. The higher the other singular values ($\sigma_2, \sigma_3, ..., \sigma_{min(T,M)}$), the less correlated all the columns are. When the correlation in the columns of the ECS matrix is low (i.e., high TMA), most tasks will have unique ranking of the machines in terms of execution speed. Alternatively, when the correlation in the columns of the ECS matrix is high (i.e., low TMA), most tasks will have the same ranking of machines in terms of execution speed performance. The TMA of the matrix is defined as the average of all the non-maximum singular values (i.e., not including σ_1) of the standard matrix:

$$TMA = \sum_{i=2}^{min(T,M)} \sigma_i/(min(T, M) - 1). \tag{8}$$

For creating the low TMA matrices, we want to have an environment where the columns of the ECS matrix are completely correlated. We do this by removing the effect of all non-maximum singular values and only retain the characteristics of the first singular value. This is equivalent to taking the rank-1 approximation of the ECS matrix (i.e., make all the non-maximum singular values to be equal to 0).

For creating the high TMA matrices, simply increasing the non-maximum singular values or decreasing the maximum singular value will typically result in matrices with negative values or result in very little increase in the TMA. Negative values in an ECS matrix are meaningless representations of execution speed of a task on a machine and are therefore undesirable. We design an iterative method to increase the components of the ECS matrix

in the directions that result in higher TMA while making sure that negative values are never introduced into the matrix. To do this, we first take the SVD of the ECS matrix A. Let u_1, u_2, u_3, etc., represent the columns of the U matrix and let v_1^T, v_2^T, v_3^T, etc., represent the rows of the V^T matrix resulting from the SVD of A. Our goal is to increase the non-maximum singular values (without introducing negative values) to get a new ECS matrix with a higher TMA. We examine how much of the matrix resulting from the product of u_2 and v_2^T we can add to A, without making any of the elements in the matrix negative. We then do this with u_3 and v_3^T, and continue to do this iteratively for all the non-maximum singular values. This allows one to increase the TMA of the environment without having any negative values in the matrix.

These procedures to create low and high TMA environments ensure that only the TMA of the environment is affected while maintaining the other characteristics of the matrix. The MPH and the TDH of the low and high TMA environments have negligible difference compared to that of the example environment. Table 3.2 shows the range of TMA values for the matrices of the example environment and also of the low and high TMA environments (that we created using our technique described above). Recall that the construction of these TMA environments are for simulation experiments to evaluate the heuristics. They are not part of the heuristics or system model.

TABLE 3.2. Range of Task-Machine Affinity (TMA) Values for the 48 Simulation Trials of the Different Environments

type of ETC	**TMA range**
example environment	0.082 to 0.091
low TMA	$< 10^{-15}$
high TMA	0.14 to 0.18

3.5.4. Obtaining an Energy Constraint. In many real-world scenarios, an annual energy budget is typically given for an HPC system. As mentioned in Section 3.2.2, we can estimate the energy constraint of the current day using a given annual energy constraint to help ensure each day uses an appropriate portion of the remaining energy from the annual budget.

For simulation purposes, we need to create an energy constraint that we can use to analyze our resource management techniques. We first run Max-Max UPT (the heuristic that provides the best utility earned from our previous work [2]) for a full 24-hour time period, disregarding the energy constraint. Based on the resource allocations generated by this heuristic, we average the total energy consumption throughout the day across 48 simulation trials and use 70% of this average value as our energy constraint. As a result, for our simulations, we used an energy constraint value of 1.11 GJ per day.

3.6. Results

3.6.1. Overview. All results shown in this section display the average over 48 simulation trials with 95% confidence interval error bars (the simulator uses two 24 core nodes on the Colorado State University ISTeC Cray HPC cluster [46]). We first discuss the performance of the heuristics in the energy-constrained environment when not using the energy filtering technique. All the heuristics used a dropping threshold of 0.5 units of utility to tolerate the oversubscription, i.e., a task is dropped if the best possible utility it can earn is lower than 0.5. We use a dropping threshold of 0.5 units of utility as it gave the best performance in our previous work [2]. When selecting a dropping threshold, we must consider the level of oversubscription of the environment in addition to the utility values of tasks. We then examine the effect of the filtering mechanism on the heuristics with a sensitivity study and

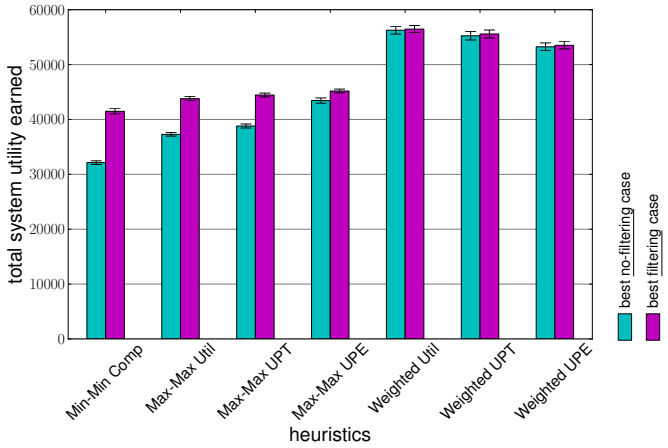
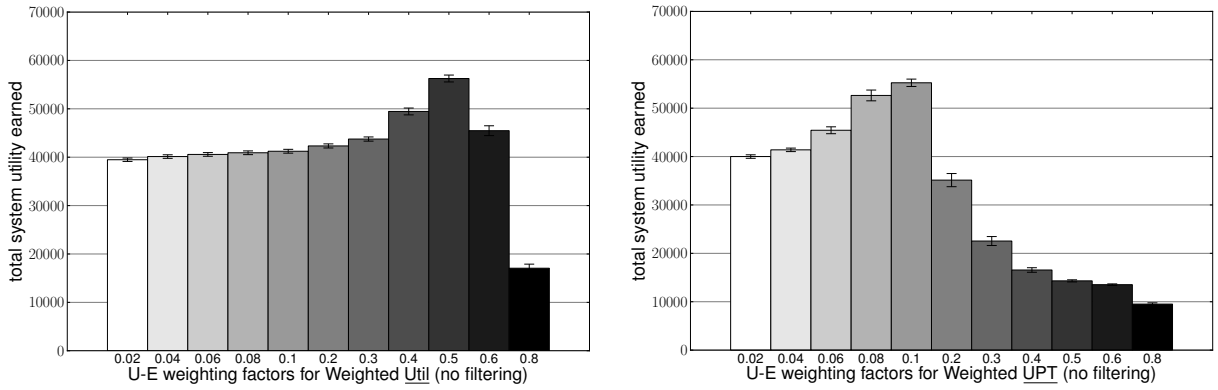


FIGURE 3.4. Total utility earned by the heuristics in the no-filtering case and their best filtering case (case with the best performing value of *energy leniency*). For the weighted heuristics, in both the cases, the best performing *U-E weighting factor* was chosen. The simulated system has 100 machines and approximately 50,000 tasks arriving in the day. These results are for the example environment. The results are averaged over 48 trials with 95% confidence intervals.

an analysis of the performance. We then analyze the performance of the heuristics in the relatively low and the high TMA environments and finally compare the best performing case for all the heuristics in the different types of environments.

3.6.2. EXAMPLE ENVIRONMENT RESULTS IN NO-FILTERING CASE. Figure 3.4 shows the total utility earned by the heuristics in the filtering and no-filtering cases. We first discuss the performance of the heuristics in the no-filtering case. Our four new heuristics outperform the heuristics from the literature (i.e., Min-Min Comp, Max-Max Util, and Max-Max UPT). Among the non-weighted heuristics, Max-Max UPE earns the highest utility even though it consumes the same amount of energy as the others. This is because the heuristic accounts for energy consumption while trying to maximize utility, and thus is able to avoid high energy-consuming allocation choices without significantly affecting the utility earned. Once the allotted energy for the day ($energy\ constraint_{day}$) is consumed, the heuristics were not allowed to map any more tasks until the following day.



(A) (B)

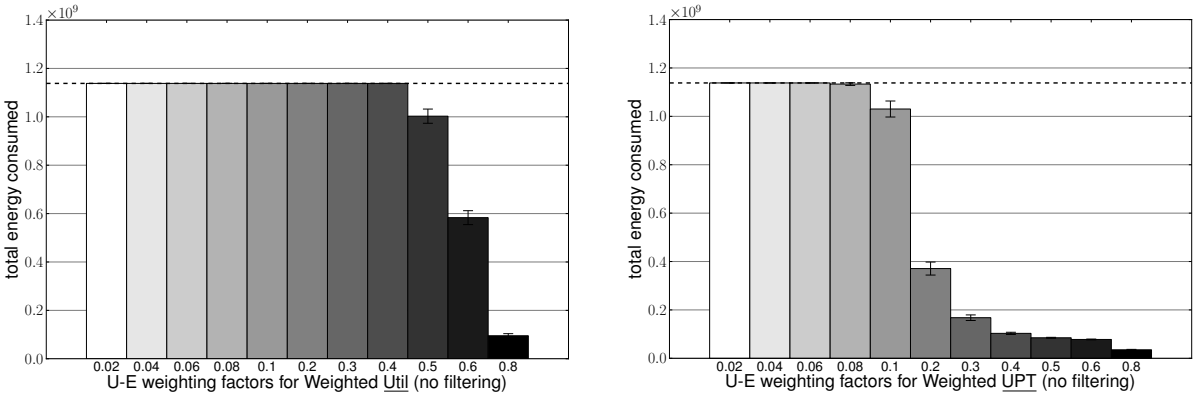
FIGURE 3.5. Tests showing the total utility earned in the no-filtering case as the *U-E weighting factor* is varied for (a) Weighted Util and (b) Weighted UPT. The simulated system has 100 machines and approximately 50,000 tasks arriving in the day. These results are for the example environment. The results are averaged over 48 trials with 95% confidence intervals.

The weighted heuristics perform much better than the other heuristics because of their ability to balance the extent to which they want to bias their allocations towards utility maximization versus energy minimization. For each of the weighted heuristics, tests were performed with different values of the *U-E weighting factor* from 0.02 to 0.8. A higher *U-E weighting factor* biases the allocation decisions more towards energy minimization. Figures 3.5a and 3.5b show the total utility earned by the Weighted Util and Weighted UPT heuristics for different values of the *U-E weighting factor*. Similarly, Figures 3.6a and 3.6b show the total energy consumption by the Weighted Util and Weighted UPT heuristics for different values of the *U-E weighting factor*. Weighted UPE showed similar trends as the Weighted UPT heuristic.

As we vary the *U-E weighting factor* from 0.02 to 0.8, the utility earned by the weighted heuristics increases and then decreases. At very low values of the *U-E weighting factor*, the energy term in the weighted expression has very little impact and the Weighted Util and Weighted UPT heuristics approaches the Max-Max Util and Max-Max UPT heuristics,

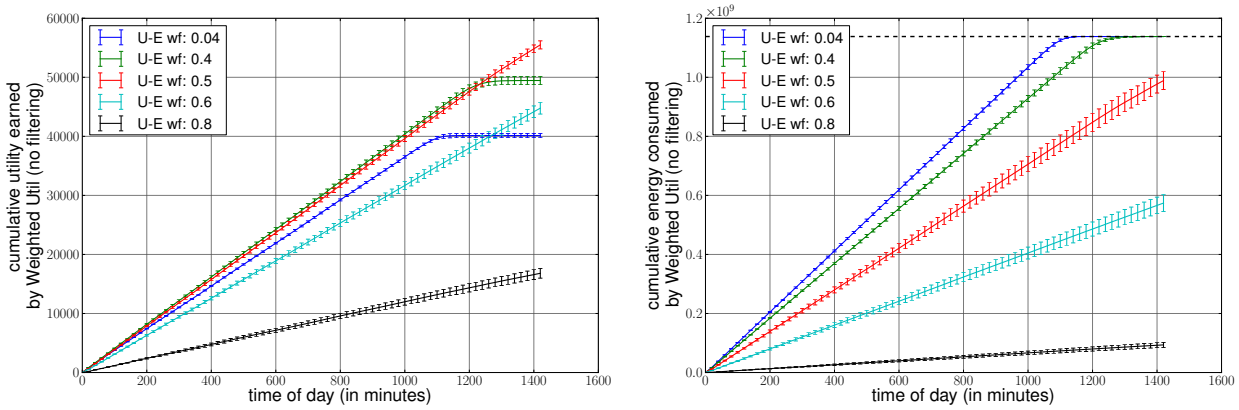
respectively. With very high values of the *U-E weighting factor*, the heuristics are too conservative in their energy expenditure and only execute tasks that consume the least energy with little regard to the utility being earned. As can be seen in Figures 3.5a and 3.5b, the best performance is obtained between these extremes. For the Weighted Util heuristic, the best performance is obtained at a *U-E weighting factor* that is larger than the best performing *U-E weighting factor* for the Weighted UPT and Weighted UPE heuristics. This is because the Weighted Util heuristic completely depends on the energy term in the weighted expression to be energy-aware. The Weighted UPE and the Weighted UPT heuristics already are able to account for energy minimization (directly or indirectly) using the first part of their weighted expression, and therefore need a smaller portion of the energy term to help make good low energy consuming choices. The energy consumption by these heuristics always hits the energy constraint for the weighting factors lower than the best performing weighting factor. Higher values of the *U-E weighting factor* further minimize energy consumption (so it unnecessarily goes below the constraint) leading to reduced performance.

To illustrate why *U-E weighting factor* of 0.5 is the best for the Weighted Util heuristic, Figures 3.7a and 3.7b show the trace of the total utility being earned and the trace of the total energy consumption for the Weighted Util heuristic in the no-filtering case for different values of the *U-E weighting factor*. For *U-E weighting factors* 0.4 and lower, the energy constraint is hit before the end of the day and therefore high utility-earning tasks that arrive after that point in time are unable to execute. This causes a drop in performance because no utility is earned from such tasks after this point. Values of *U-E weighting factor* higher than 0.5 are too conservative in energy consumption and have too much energy left over at the end of the day.



(A) (B)

FIGURE 3.6. Tests showing the total energy consumption in the no-filtering case as the *U-E weighting factor* is varied for (a) Weighted Util and (b) Weighted UPT. The dashed horizontal line shows the energy constraint of the system. The simulated system has 100 machines and approximately 50,000 tasks arriving in the day. These results are for the example environment. The results are averaged over 48 trials with 95% confidence intervals.



(A) (B)

FIGURE 3.7. Traces of (a) the cumulative utility earned and (b) the cumulative energy consumption for the Weighted Util heuristic in the no-filtering case throughout the day at 20 minute intervals at different *U-E weighting factors*. The dashed horizontal line in (b) shows the energy constraint of the system. The simulated system has 100 machines and approximately 50,000 tasks arriving in the day. These results are for the example environment. The results are averaged over 48 trials with 95% confidence intervals.

These results indicate that for energy-constrained environments it is best to use heuristics that consider energy, rather than heuristics that try to solely optimize for maximizing utility.

In the no-filtering case, Weighted Util is approximately 28% better than Max-Max UPE, and Max-Max UPE is approximately 11% better than Max-Max UPT (the best-performing utility maximization heuristic from the literature [2]).

3.6.3. Example Environment Results with Energy Filtering. We now examine the effect of the energy filtering mechanism on the batch-mode heuristics. The extent to which energy filtering is performed is controlled by the *energy leniency* term (see Equation 7). A higher value for the *energy leniency* would result in a higher value of the *task budget* and would therefore let more allocation choices pass through the filter. Alternatively, a lower value of *energy leniency* would let fewer allocations pass through the filter. Not using filtering implies an *energy leniency* value of infinity. We performed a sensitivity test for all the heuristics by varying the value of *energy leniency* from 0.3 to 4.0, and compared the performance with the no-filtering case. We first analyze the performance of the non-weighted heuristics in Section 3.6.3.1, and then examine the weighted heuristics in Section 3.6.3.2.

3.6.3.1. *Non-weighted Heuristics.* Figures 3.8a and 3.8b show the effect of varying the value of *energy leniency* on the total utility earned by the Max-Max UPT and the Max-Max UPE heuristics, respectively. Figure 3.9 shows the energy consumption of the Max-Max UPE heuristic as the *energy leniency* value is varied. Sensitivity tests of the utility earned for the Min-Min Comp and Max-Max Util heuristics show trends similar to that of the Max-Max UPT heuristic, while the sensitivity tests of the energy consumed for Min-Min Comp, Max-Max Util, and Max-Max UPT showed trends similar to Max-Max UPE.

In general, for the non-weighted heuristics, as we increase the value of *energy leniency* from 0.3, the utility earned increases and then decreases as we approach the no-filtering case. All of these heuristics benefit from the filtering operation. The best-performing case for Min-Min Comp, Max-Max Util, and Max-Max UPT occurs at an *energy leniency* of 0.75,

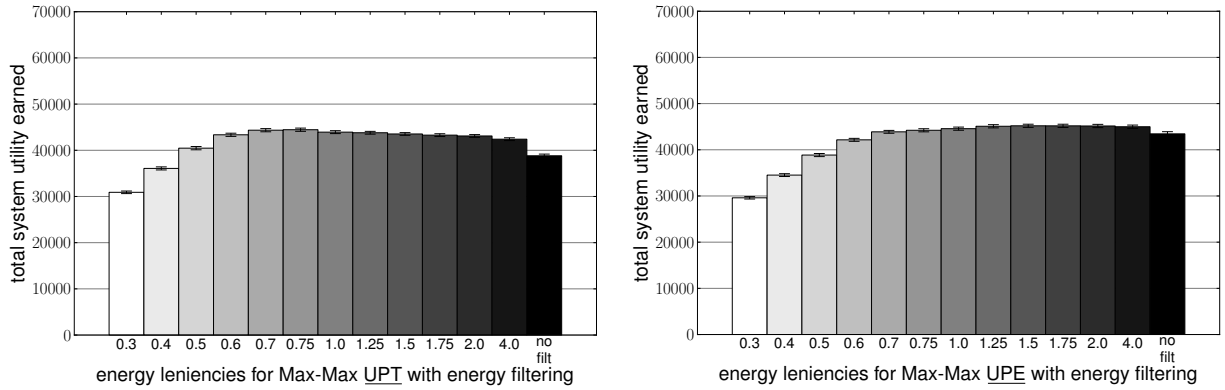


FIGURE 3.8. Sensitivity tests showing the total utility earned as the *energy leniency* is varied for (a) Max-Max UPT and (b) Max-Max UPE. The simulated system has 100 machines and approximately 50,000 tasks arriving in the day. These results are for the example environment. The results are averaged over 48 trials with 95% confidence intervals.

whereas the Max-Max UPE heuristic performance peaks at an *energy leniency* of 1.5. We observe that the performance benefit for the Max-Max UPE heuristic is less sensitive to the value of *energy leniency*, especially in the range 1.0 to 4.0. The drop in performance for this heuristic in the no-filtering case (compared to its best performance case) is less substantial than the similar difference for the other heuristics. This is because the Max-Max UPE heuristic already accounts for energy consumption, reducing the benefits associated with the energy filter. Therefore, the best-performing case of *energy leniency* for this heuristic is at a higher value of *energy leniency* than the best-performing case for the other heuristics. The other heuristics require a stricter filtering technique to incorporate energy consumption in allocation choices, therefore they require lower values of *energy leniency* to obtain the best results, because energy is not considered otherwise.

For the non-weighted heuristics, when we use *energy leniency* values from 0.3 to 0.6, the filtering is so strict that it prevents the heuristic from using all of the available energy that was budgeted for the day. Not being able to use all of the budgeted energy results in

fewer tasks being executed and therefore a drop in the total utility earned throughout the day. Alternatively, when using high values of *energy leniency* (and the no-filtering case), all heuristics use all of the day's budgeted energy early in the day and thus are unable to execute tasks that arrive in the later part of the day. We are able to observe this using trace charts that show the gain in total utility and increase in total energy consumption.

Figures 3.10a and 3.10b show the utility trace for the Max-Max UPT and the Max-Max UPE heuristics, respectively. Figures 3.11a and 3.11b show the energy trace for the Max-Max UPT and the Max-Max UPE heuristics, respectively. For the no-filtering case, we see that the system uses all of the available energy for the day in the early part of the day, and then all future tasks are unable to execute and are dropped from the system earning no utility. The no-filtering case for the Max-Max UPE heuristic uses all available energy that was budgeted for the day slightly later (approximately three hours) than the Max-Max UPT heuristic because the heuristic considers energy at each mapping event throughout the day. The slope of its no-filtering energy consumption trace is less steep than the slope of the similar trace for the Max-Max UPT heuristic. As a result, Max-Max UPE is able to execute more tasks and earn higher utility.

The energy trace charts show the adaptive ability of the filtering technique. Recall the task budget is dependent on the *aggregate time remaining* and the *energy remaining*. When comparing low values of *energy leniency* to high values of *energy leniency*, the *energy remaining* will be similar at the beginning of the day, but later in the day, there will be more energy remaining for low values compared to the lower energy remaining for higher values. Therefore, because the *task budget* will change with the energy remaining, it will become larger when there is more energy remaining in the day and smaller when there is less energy remaining in the day. For example, the slope increases for the *energy leniency* line of

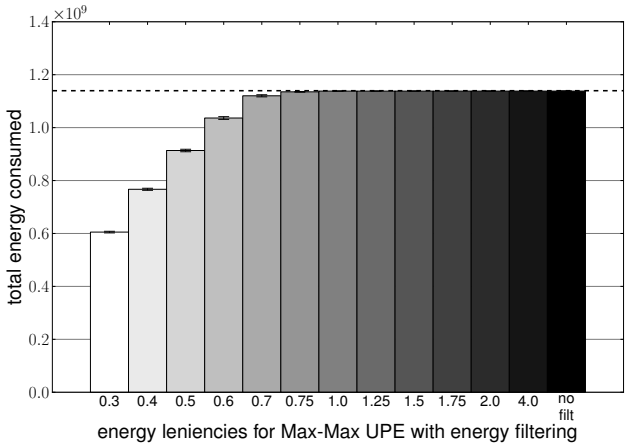
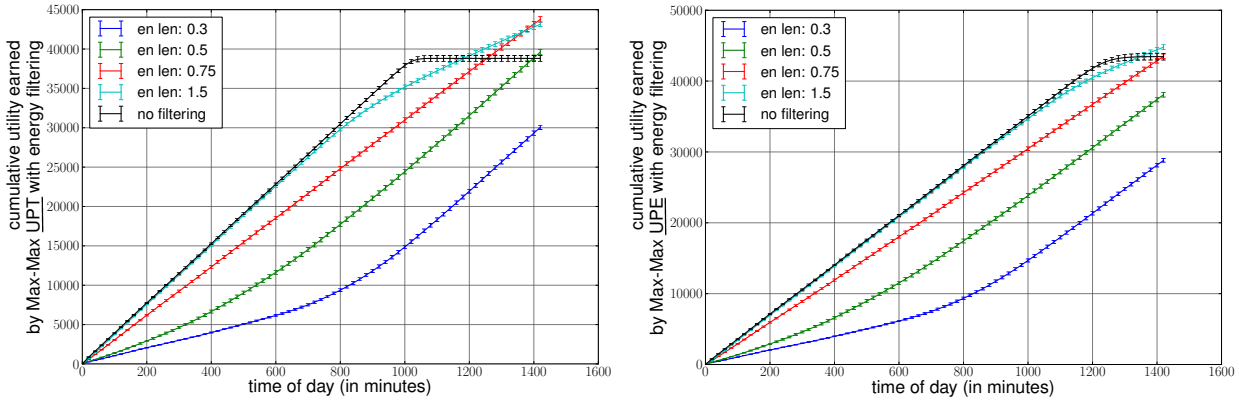


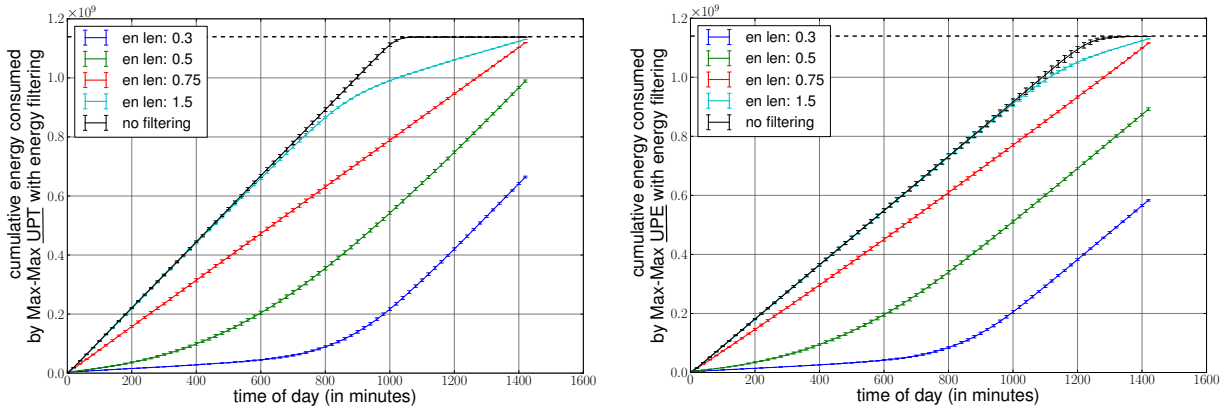
FIGURE 3.9. Sensitivity tests showing the total energy consumed as the *energy leniency* is varied for the Max-Max UPE heuristic. The dashed horizontal line shows the energy constraint of the system. The simulated system has 100 machines and approximately 50,000 tasks arriving in the day. These results are for the example environment. The results are averaged over 48 trials with 95% confidence intervals.



(A) (B)

FIGURE 3.10. Traces of the cumulative utility earned throughout the day at 20 minute intervals as the *energy leniency* (en len) is varied for the (a) Max-Max UPT and (b) Max-Max UPE heuristics. The simulated system has 100 machines and approximately 50,000 tasks arriving in the day. These results are for the example environment. The results are averaged over 48 trials with 95% confidence intervals.

0.3 during the day in Figures 3.11a and 3.11b. Similarly, with high values of *energy leniency*, the filter eventually adapts to lower its value of *task budget*. This is shown by the decrease in slope for the 1.5 *energy leniency* line in Figures 3.11a and 3.11b.



(A) (B)

FIGURE 3.11. Traces of the cumulative energy consumed throughout the day at 20 minute intervals as the *energy leniency* (en len) is varied for the (a) Max-Max UPT and (b) Max-Max UPE heuristics. The dashed horizontal line shows the energy constraint of the system. The simulated system has 100 machines and approximately 50,000 tasks arriving in the day. These results are for the example environment. The results are averaged over 48 trials with 95% confidence intervals.

The best performance for each of the non-weighted heuristics comes at an appropriate *energy leniency* that allows the total energy consumption of the heuristic to hit the energy constraint of the day right at the end of the day, saving enough energy for any high-utility earning tasks that may arrive at later parts in the day. Higher values of *energy leniency* (above 1.5) result in the energy constraint being hit in the earlier part of the day, while lower values of *energy leniency* can result in a strict environment that prevents the consumption of all of the energy budgeted for the day. Therefore, in energy-constrained environments, the best performance is obtained by permitting allocation choices with a fair-share of energy so that the total energy consumption for the day hits the energy constraint right at the end of the day. By doing so relatively low utility-earning tasks that arrive early in the day do not consume energy that could be used by relatively higher utility-earning tasks arriving later in the day. If the energy consumption is not regulated, then the allocations in the earlier part

of the day can consume too much energy preventing task executions later in the day. Our energy filtering technique gives this ability to these heuristics.

Among the non-weighted heuristics, Max-Max UPE performs the best and its performance is the least sensitive to the value of *energy leniency*. These are because this heuristic accounts for the energy consumed to earn each unit of utility. The performance of all the non-weighted heuristics improves because of the energy filtering technique. When designing an energy filter for a heuristic in an oversubscribed environment, the best performance is likely to be obtained when the level of filtering is adjusted to distribute the consumption of the energy throughout the day and meet the constraint right at the end of the day. This can be used to design filters for such heuristics in energy-constrained environments to maximize performance.

As mentioned in Section 3.5.4, we set the energy constraint to approximately 70% of the average total energy consumed by an unconstrained execution of the Max-Max UPT heuristic. If the energy constraint was set to a lower percentage than 70%, then the relative performance of the heuristics would be exaggerated, further highlighting the benefit of the filtering and weighting techniques. On the contrary, having a very high energy constraint value, would reduce the relative performance benefit of these techniques. Changing the energy constraint essentially changes the time at which the trace charts of the no filtering-no weighting case start to plateau. This directly impacts the total utility they can earn. The less tighter the energy constraint, the later the trace charts start to plateau, and as a result, the higher the total utility they earn.

3.6.3.2. *Weighted Heuristics.* The weighted heuristics already have the ability to tune their energy consumption throughout the day and therefore they do not benefit from the energy filtering technique, as shown in Figure 3.4. Figure 3.12 shows the utility earned

by the Weighted Util heuristic for different combinations of *U-E weighting factor* and *energy leniency*. As seen in Figure 3.6a, even in the no-filtering case, the Weighted Util heuristic did not consume the total energy budgeted for the day with *U-E weighting factors* 0.5 and above. Therefore, adding energy filtering (that may further limit the energy consumption) to these cases does not help to improve the performance of the heuristic in comparison to the no-filtering case. Using moderate values of *energy leniency* helps in cases where we use lower *U-E weighting factors*, because in these cases, the weighting factor alone is unable to accomplish the desired level of energy minimization. At the best performing *U-E weighting factor* (i.e., 0.5 for Weighted Util), the no-filtering case performs just as well as the best performing *energy leniency* cases.

Both, the filtering technique and the weighting technique, have the ability to regulate the energy consumption throughout the day and allow the energy constraint to be hit only at the end of the day, but the weighting technique performs better than the filtering technique. We now analyze why the best performing *U-E weighting factor* (without any energy filtering, i.e., *energy leniency* of infinity) performs better than the best performing energy filtering case (without using any weighting, i.e., *U-E weighting factor* $= 0$). To study this difference, we plot the utility and energy trace charts of the Weighted Util heuristic for the following three scenarios:

(1) no filtering (*energy leniency* $= \infty$) and no weighting (*U-E weighting factor* $= 0$),

(2) best filtering case (*energy leniency* $= 0.75$) and no weighting, and

(3) no filtering and best weighting case (*U-E weighting factor* $= 0.5$).

These trace charts are shown in Figures 3.13a and 3.13b. Recall that without the weighting i.e., a *U-E weighting factor* of 0, the weighted heuristics reduce to their non-weighted counterparts (e.g. Weighted Util becomes Max-Max Util).

The weighting case outperforms the filtering case because of two reasons. Each of these reasons can be explained by examining the trace up to the point where the no filtering-no weighting case hits the energy constraint at approximately 1000 minutes. Recall that the no filtering-no weighting case is only attempting to maximize utility with no regard to energy. The first reason why the weighting performs better than the filtering is because the filtering removes allocation choices that consume more energy than the fair-share *task budget* (of a mapping event) without considering the utility that that allocation choice may earn. This causes the filtering case to avoid certain high energy consuming (but high utility earning) allocation choices to execute. This can be seen by the lower values of utility being earned by the heuristic in the filtering case compared to the no filtering-no weighting case up to 1000 minutes. The weighting case does not have this problem as it is able to rank choices in terms of both the utility they earn and the energy they consume. So, if an allocation choice consumes high energy but proportionally earns high utility then this allocation choice may be a viable option in the weighting case. Weighting, if tuned correctly, allows the heuristic to balance the required amount of energy minimization versus utility maximization. The second reason why the weighting case performs better is because the weighting case biases decisions to pick low energy consuming allocation choices, and in our environment this also leads to minimization of execution time. Therefore, Weighted Util gets the ability to make allocations that behave as utility-per-time. Because of the minimization of execution time, we observed that the weighting case was able to complete many more tasks compared to the no filtering-no weighting case. This causes the heuristic's weighting case to earn higher utility than the unconstrained no filtering-no weighting case even in the region up to 1000 minutes.

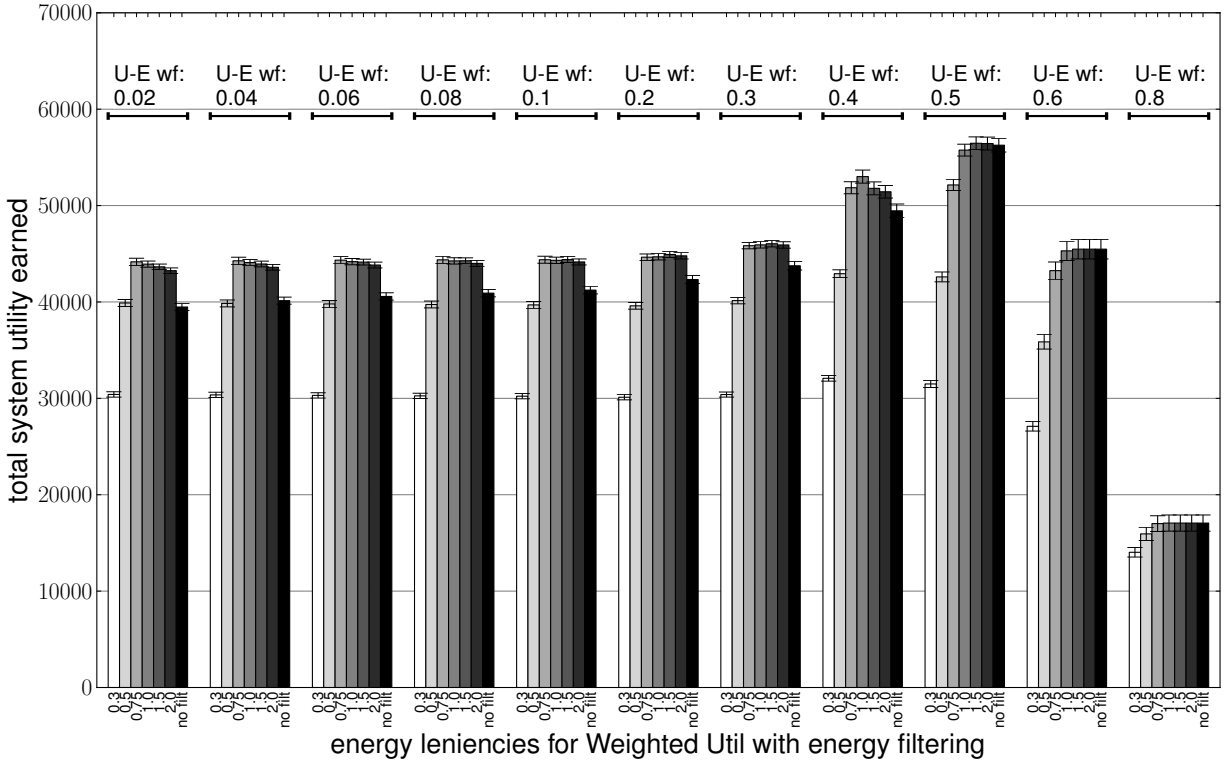
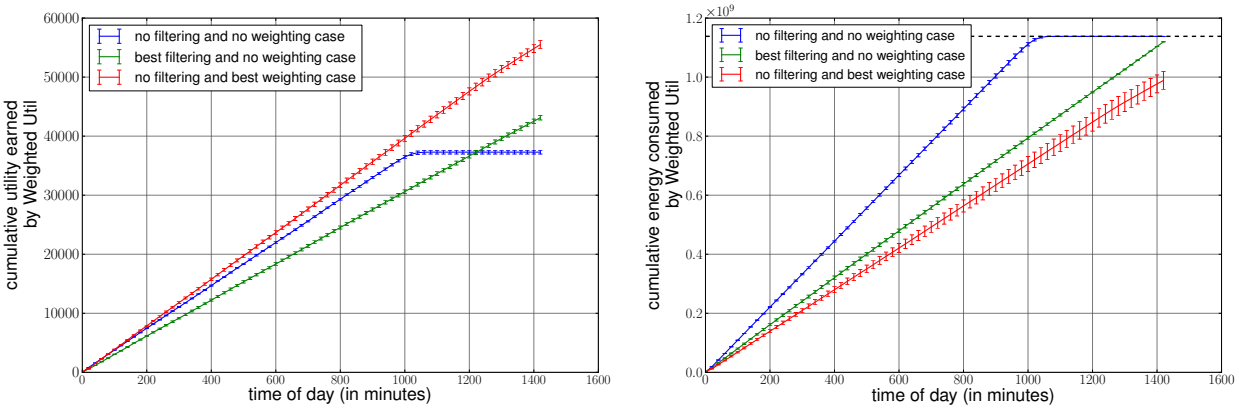


FIGURE 3.12. Sensitivity tests showing the total utility earned for different combinations of *U-E weighting factor* (U-E wf) and *energy leniency* for the Weighted Util heuristic. The simulated system has 100 machines and approximately 50,000 tasks arriving in the day. These results are for the example environment. The results are averaged over 48 trials with 95% confidence intervals.

The weighting and filtering techniques both allow a heuristic to regulate its energy consumption throughout the day and permit task executions in the later part of the day that help them to earn more utility than a case that does not use either weighting or filtering. Filtering does not attempt to minimize energy consumption, it only tries to prune high energy consuming choices. It makes its filtering mechanism stricter or more lenient if the energy consumption rate is higher or lower than what it should ideally be, respectively. As opposed to this, weighting works by attempting to minimize energy consumption right from the start of the allocation process. It picks allocation choices that have a good balance of earning utility versus consuming energy.



(A) (B)

FIGURE 3.13. Traces of (a) the cumulative utility earned and (b) the cumulative energy consumption throughout the day at 20 minute intervals of different cases of using the best/not using energy filtering and/or weighting for the Weighted Util heuristic. The dashed horizontal line in (b) shows the energy constraint of the system. The simulated system has 100 machines and approximately 50,000 tasks arriving in the day. These results are for the example environment. The results are averaged over 48 trials with 95% confidence intervals.

Figure 3.4 shows the total utility earned by the heuristics in the no-filtering case and their best *energy leniency* case. The non-weighted heuristics have a significant performance increase with the energy filtering because it allows them to control their energy expenditure throughout the day. The weighted heuristics already have the ability to control their energy consumption and therefore do not have any increase in performance when using energy filtering.

As mentioned in Section 3.5.4, we set the energy constraint to approximately 70% of the average total energy consumed by an unconstrained execution of the Max-Max UPT heuristic. Changing the energy constraint essentially changes the time at which the trace charts of the no filtering and no weighting case start to plateau. This directly impacts the total utility they can earn. The looser the energy constraint, the later the trace charts start to plateau, and as a result, the higher the total utility they earn. If the energy constraint was

set to a lower percentage than 70%, then the relative performance of the heuristics would be exaggerated, further highlighting the benefit of the filtering and weighting techniques. On the contrary, having a very high energy constraint value (loose constraint), would reduce the relative performance benefit of these techniques.

3.6.4. Low and High TMA Environment Results with Filtering. We ran similar experiments as mentioned thus far for all the heuristics with the low and high TMA environments. These environments have the same set of task utility functions, task arrival times, oversubscription levels, dropping threshold, overall aggregate performance of the machines and tasks with similar values of MPH and TDH compared to the example environment. The only difference is the TMA of the environment, and that affects the uniqueness by which certain task types execute faster on certain machine types. In the low TMA environment, all tasks have the same ranking of machines in terms of execution time, whereas in the high TMA environment, most tasks have unique ranking of the machines in terms of execution time performance. We ran parameter tuning tests to find the best performing *energy leniency* case for the non-weighted heuristics and the best performing *energy leniency* and *U-E weighting factor* case for the weighted heuristics. Here, we summarize the conclusions obtained from these tests. The actual results of the tests are omitted from the draft for brevity.

For the non-weighted heuristics in the low TMA environment, the best performance was obtained at a higher value of *energy leniency*. This was because in the low TMA environments, as all tasks have the same ranking of the machine types in terms of execution time, fewer tasks get to execute on the machines where they have better execution times. On average, this results in longer task execution times than that in the example environment. Therefore, the *average execution time* estimate is not as accurate as it was in the example

environment. To compensate for this, the best performance is obtained at higher values of *energy leniency*. See Equation 7 for understanding how an increase in *energy leniency* helps to make up for the lower value of the *average execution time* estimate.

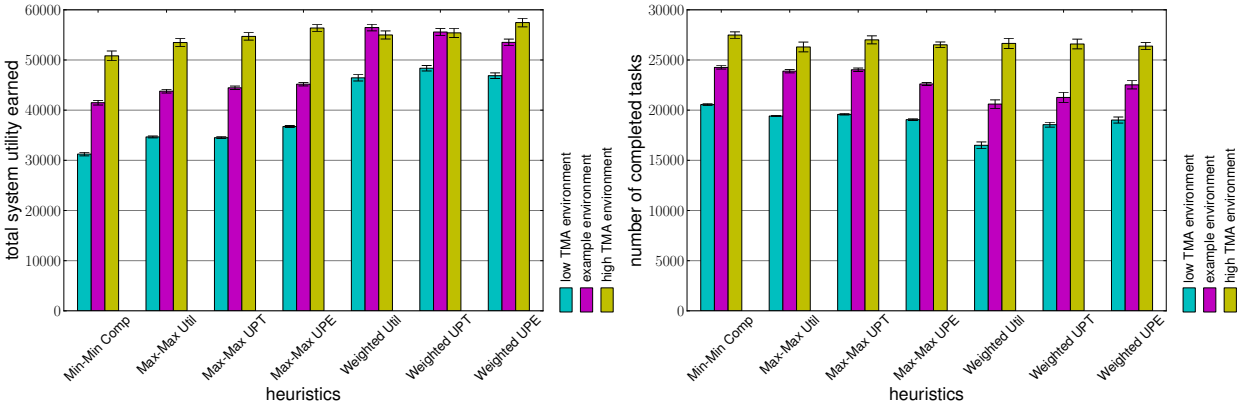
For the weighted heuristics in the low TMA environment, the best performance was obtained at lower values of the *U-E weighting factor* (i.e., more weighting towards utility term) compared to the *U-E weighting factors* for the best case in the example environment. The energy term of the weighted expression does not account for how oversubscribed the machine is. In a low TMA environment, all tasks would have the same ranking of machine-P-state choices in terms of energy consumption. Assigning all tasks to these energy efficient machines would oversubscribe them resulting in low utility being earned from the tasks. By having a lower value for the *U-E weighting factor*, the preferred allocation choice would be less biased towards the minimization of energy consumption, therefore allowing for a better load balance in a low TMA environment.

In contrast, in high TMA environments, the non-weighted heuristics performed their best at lower values of *energy leniency*. In such environments, different tasks have different ranking of the machine types in terms of execution time and this makes it easier for tasks to be assigned to their best execution time machine type. As a result, the estimate of the *average execution time* (calculated based on ETC values) is not as accurate as it was in the example environment. The best performance is obtained at lower values of *energy leniency* as that helps to compensate for the higher value of the *average execution time* estimate.

For the weighted heuristics, in high TMA environments, the best performance is obtained at higher values of *U-E weighting factor* (i.e., more weighting towards energy term). This is because in the high TMA environment, different tasks would generally have different machine-P-state choices that execute fast and consume less energy. Therefore, biasing the

scheduling decisions more towards the minimum energy consumption choices provides automatic load balancing across the machine types and assigns tasks to the machine types that execute them the best (i.e., quickest and least energy consuming).

Figure 3.14a shows the utility earned by the best performing case of each of the heuristics in the low and high TMA environments in comparison to the best performance obtained in the example environment. The overall trend is that heuristics earn less utility in the low TMA environment and more utility in the high TMA environment compared to the example environment. This is because in the low TMA environment, all tasks have the same machine type as their best execution time machine type and as a result fewer tasks get to execute on the machines of this type. This leads to longer execution times on average for the tasks and results in fewer tasks being pushed through the system during the day. Alternatively, in the high TMA environment, different task types execute fastest on different machine types, and therefore, assigning tasks to their best execution time machine type is feasible as it implicitly provides load balancing. In such an environment, the average execution time of the tasks is lower and more tasks complete execution during the day earning higher utility overall. In high TMA environments, resource allocation decisions are easier to make (as minimizing execution time provides some load balancing) and therefore the performance of all the heuristics is quite similar. Although, the weighted heuristics still perform slightly better than their non-weighted counterparts. Figure 3.14b shows that compared to the example environment, the number of tasks completed for the low TMA environment is significantly fewer while the number of tasks completed for the high TMA environment is significantly greater. The average execution time of a task across the machines (taken from the ETC) is similar for the low and example environment and slightly higher for the high TMA environment. We still get better performance in the high TMA environment from



(A) (B)

FIGURE 3.14. (a) Total utility earned and (b) Total number of completed tasks by the best performing cases for all the heuristics with energy filtering in the three types of environments: low TMA, example, and high TMA. The simulated system has 100 machines and approximately 50,000 tasks arriving in the day. The results are averaged over 48 trials with 95% confidence intervals.

all the heuristics because of the diversity in the tasks' execution time performance across machine types. The number of dropped tasks can be obtained by subtracting the numbers in Figure 3.14b from 50,000 (which is approximately the total number of tasks arriving into the system).

3.7. CONCLUSIONS

In this study, the problem of energy-constrained utility maximization is address. An oversubscribed heterogeneous computing system where tasks arrive dynamically and are mapped to machines for execution is modeled. The system model is designed based on types of systems of interest to DoD/DOE. A heuristic's performance in the system is measured in terms of the total utility that it could earn from task completions. Four heuristics were designed for this problem and their performance was compared with other heuristics adapted from previous work. Additionally an energy filtering technique was integrated into the environment.

It was shown that in an energy-constrained environment energy-aware heuristics earns more utility than heuristics that only optimize for utility. The new energy filter helps to improve the performance of all the non-weighted heuristics by distributing the consumption of the budgeted energy throughout the day. The energy filtering technique adapts to the energy remaining in the system and accordingly budgets the permitted energy for a task's execution. For the non-weighted heuristics, the best performance from the filtering is obtained for all heuristics at a level of filtering that distributes energy consumption approximately equally throughout the day and meets the energy constraint right at the end of the day. This can be used to guide the design of heuristics and filtering techniques in oversubscribed heterogeneous computing environments.

The weighted heuristics have the ability to minimize energy consumption throughout the day and can be tuned so that their energy consumption meets the energy constraint right at the end of the day. These heuristics outperform their non-weighted counterparts (even when they use energy filtering). This is because, filtering considers energy regardless of utility while weighting considers them together. The filtering removes certain high energy-consuming allocation choices, but among the remaining choices it simply picks the one that maximizes the objective of the heuristic. Alternatively, the weighted heuristics rank all allocation choices by accounting for both utility and energy and have the ability to balance the degree to which they are energy-aware. As a result, these heuristics perform much better than the non-weighted heuristics (even when they use filtering).

In the low and high TMA environments, all the heuristics earn lower and higher utility overall, respectively. This is because the higher the TMA of the environment, the higher the number of tasks that can be completed because more tasks can be assigned to the machine on which they have the fastest execution time. This is because assigning tasks to their best

execution time machine implicitly balances the load. Also, in high TMA environments, it was observed that mapping decisions are easier to make and most heuristics perform similarly. Possible directions for future research are presented in Chapter 9.

CHAPTER 4

Comparison of Energy-Constrained Resource Allocation Heuristics under Different Task Management Environments[1]

4.1. Introduction

The need to solve applications of higher complexity with greater accuracy combined with the need for faster execution from high-performance computing (*HPC*) systems is resulting in higher energy consumption and costs to operate these systems. A recent National Resources Defense Council report showed that data centers in the U.S. consumed an estimated 91 billion kWh in 2013 (that is double the amount of electricity consumed by all of the households in New York City) and are on track to reach 140 billion kWh by 2020 [52]. Some data centers are now unable to increase their computing performance due to physical limitations on the availability of energy. For example, in 2010, Morgan Stanley, a global financial services firm based in New York, was physically unable to draw the energy needed to run a data center in Manhattan [41]. Many HPC systems are now being forced to execute with constraints on the amount of energy they can consume. The issue of increased energy consumption

[1]This work appeared in [51]. This work was supported by Oak Ridge National Laboratory and their Extreme Scale Systems Center under subcontract numbers 4000094858 and 4000108022, and by a National Science Foundation Graduate Research Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research also used resources of the Extreme Scale Systems Center at Oak Ridge National Laboratory as well as the CSU ISTeC Cray System supported by NSF Grant CNS-0923386.This manuscript has been supported by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

is estimated to significantly worsen as we approach exascale systems. As a result, there is a growing concern regarding energy needed to operate these systems (e.g., [37, 38]) and it is becoming increasingly important for system administrators to adopt energy-efficient workload execution policies.

This research builds on prior work that developed energy-aware resource management techniques with the goal of maximizing the performance of a workload executing on an energy-constrained heterogeneous HPC system [36]. In this work, the performance of the resource management techniques in *task management environments* that differ in their policies of which tasks and machines can be considered by the scheduling techniques are analyzed. Specifically, a queued and a polled environment are studied and contrasted. Also, an enhancement is designed for the energy filter technique (introduced in [36]) by removing the need to determine its parameters empirically and making it adaptive by using current system information. Such a filtering technique can be deployed directly into any environment without the need to be tuned off-line. The goal of the filtering technique is to ignore allocation choices that use more energy than an estimated fair-share. This improves the distribution of the budgeted energy across the constrained time period.

A compute facility and workload of interest to the Extreme Scale Systems Center (*ESSC*) at Oak Ridge National Laboratory (*ORNL*) is modeled. The ESSC is a joint venture between the United States Department of Defense (*DoD*) and Department of Energy (*DOE*) to perform research and deliver tools, software, and technologies that can be integrated, deployed, and used in both DoD and DOE environments. This system incorporates heterogeneous compute resources that utilize a mix of different machines to execute workloads with diverse computational requirements. In such an environment, each task typically has different execution time and energy consumption characteristics when executed on different

machines. Machines are modeled to have different ACPI performance states (P-states) in which tasks can execute [53].

Each task in the system has a monotonically-decreasing utility function associated with it that represents the task's *utility* (or value) as a function of the task's completion time. The system performance is measured in terms of *total utility earned*, which is the sum of utility earned by all completed tasks [2]. The goal for resource management techniques in this environment is to maximize the amount of utility earned during a period of time that has a constraint on the amount of energy that can be consumed. To keep simulations tractable, the time period of a day is considered, but one could use any length of time (e.g., six hours, one month, one year). The performance of heuristics adapted from Chapters 2 and 3 with different First Come First Served (*FCFS*) heuristics in different task management environments are compared and analyzed.

In summary, this chapter makes the following contributions: (a) the design of a novel adaptive energy filtering mechanism that can be readily deployed into any environment, (b) a comparative analysis of the advantages and disadvantages of a polled task management environment that can be used in HPC environments, and (c) a comparison of multiple FCFS heuristics that are typically used in real schedulers with smarter heuristics that can improve system performance.

The remainder of this chapter is organized as follows. The next section discusses the system model and problem. Section 4.3 describes the task management environments and the resource management techniques. The simulation setup is detailed in Section 4.4. Section 4.5 discusses and analyzes the experimental results. An overview of related work is provided in Section 4.6. The chapter comes to a close with the conclusions in Section 4.7.

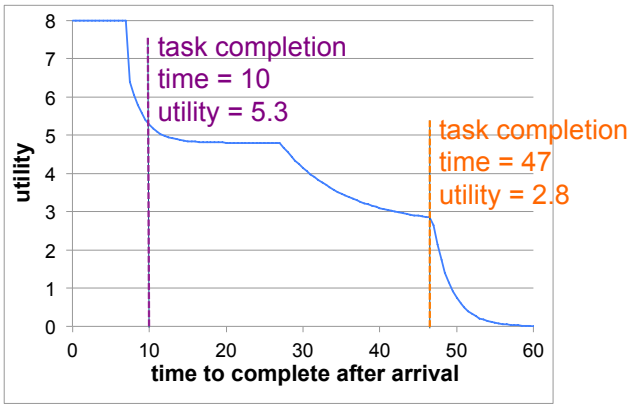


FIGURE 4.1. A sample utility function showing the utility earned at two different completion times.

4.2. PROBLEM DESCRIPTION

4.2.1. SYSTEM MODEL. In this study, the system model is similar to the model described in [36]. We model a dynamic system where tasks arrive throughout the day and a resource manager maps the tasks to machines for execution. We consider an *oversubscribed* environment, i.e., the incoming workload exceeds the capacity of the computing system. The workload and computing system we model are based on the interests of the ESSC. Each task in the system has an associated utility function (introduced in [2]). The utility function of a task is a monotonically-decreasing function that represents the "value" of completing that task at different times. Figure 4.1 shows an example utility function for a task and highlights the utility that will be earned when the task is completed at two different times. The utility function of a task is assumed to be set by the user in collaboration with the system owner. The utility functions are described by three parameters: priority, urgency, and utility class [2]. *Priority* controls the overall importance of the task by setting the task's maximum (i.e., starting) utility value. *Urgency* sets the overall rate of decay for the utility function. The *utility class* allows the shape of the utility function to be modified by partitioning it into intervals and specifying shape modifiers for each of the intervals.

Our computing system environment consists of heterogeneous machines, where each machine belongs to a specific *machine type* (rather than a single large monolithic system, such as Titan [43]). Machines belonging to different machine types may differ in their microarchitectures, memory modules, and other system components. We model the machines to contain CPUs with dynamic voltage and frequency scaling ($DVFS$) enabled to utilize different ACPI performance states (*P-states*) that offer a trade-off between execution time and power consumption. We group tasks with similar execution time and power characteristics into *task types*. Tasks belonging to different task types may differ in characteristics such as computational intensity, memory intensity, I/O intensity, and memory access pattern. The application (task) developer specifies which task type the application falls into. The type of a task is not related to the utility function of the task. Because the system is heterogeneous, machine type A may be faster (or more energy-efficient) than machine type B for certain task types but slower (or less energy-efficient) for others. We model *general-purpose* machine types and *special-purpose* machine types in our heterogeneous system [45]. The special-purpose machine types execute certain special-purpose task types much faster than the general-purpose machine types, although they may be incapable of executing the other task types.

We assume that for a task of type i on a machine of type j running in P-state k, we are given the Estimated Time to Compute ($ETC(i,j,k)$) and the Average Power Consumption ($APC(i,j,k)$). It is common in the resource management literature to assume the availability of this information based on historical data or experiments [11–15, 44, 16]. The APC incorporates both the static power (not affected by the P-state of the task) and the dynamic power (different for different P-states). We can compute the Estimated Energy Consumption ($EEC(i,j,k)$) by taking the product of execution time and average power consumption, i.e.,

$EEC(i,j,k) = ETC(i,j,k) \times APC(i,j,k)$. ETC and APC values for the ESSC environment are not available to researchers. Therefore, for the simulation study conducted in this chapter, we synthetically create ETC and APC matrices based on recommendations provided by ESSC and based on general trends of the workloads.

Tasks are assumed to be independent (they do not require inter-task communication) and can execute concurrently (each on a single machine, possibly with parallel threads). This is typical of many environments, such as [46]. We do not allow the preemption of tasks, i.e., once a task starts execution, it must execute until completion.

4.2.2. Problem Statement. With tasks dynamically arriving, the scheduler does not know the arrival time, type, or utility function of the next task. *The goal of the scheduler is to maximize the total utility that can be earned from completing tasks during a given period of time while satisfying an energy constraint (E) for that time period.* We use the duration of a day to keep the simulation time tractable. Instead of one day we could base our constraint on any interval of time (e.g., two hours, six months, a year). For ESSC, constraints on power (energy per time) are not a concern. As the system modeled is oversubscribed, the machines are never turned off, and therefore, the fraction of static versus dynamic power is not relevant.

4.3. Resource Management

4.3.1. Overview. Heuristics are commonly used to solve task to machine scheduling problems that have been shown to be NP-hard [17].A *mapping event* occurs any time a scheduling decision has to be made. We use *batch-mode heuristics* that trigger mapping events after fixed time durations (one minute in our environment) after the previous mapping event completes [9, 2].

During a mapping event, three decision processes are executed. The first operation drops tasks that have low potential utility at the current time to allow the system to better tolerate high oversubcription scenarios. The second operation is an energy filtering technique. We adapt the technique from our work in [36] to control the energy expenditure by preventing tasks from using more than their "fair-share" of energy. We enhance the energy filtering technique to use more system information and enable it to automatically adjust its level of energy filtering without the need for any parameter tuning. The final operation during a mapping event does the actual mapping of tasks to machines in certain P-states using some heuristic approach.

We study two different task management environments: queued and polled. In the *queued* environment, each of the machines has a queue of tasks that it will execute in the queue order. The task that is next-in-line for execution on a machine is referred to as the *pending task*. All other tasks that are queued for the machines are said to be in the *virtual queues* of the scheduler. Figure 4.2a shows the state of a small example system prior to a mapping event with four machines and executing tasks, tasks in the pending slots, the virtual queues of the scheduler, and the tasks that have arrived since the last mapping event. At a mapping event in the queued environment, the batch-mode heuristics make scheduling decisions for a set of tasks comprising those that have arrived since the last mapping event and the ones that are currently in the virtual queues. This set of tasks is called the *mappable tasks set*. The batch-mode heuristics are not allowed to remap the pending tasks so that the machines do not idle if the currently executing tasks complete while the heuristic is executing. As we do not allow preemption, the currently executing tasks cannot be interrupted and therefore are not available for mapping. We refer to the pending and the currently executing tasks as the *unmovable tasks*. As there are queues for the machines, the heuristics consider all

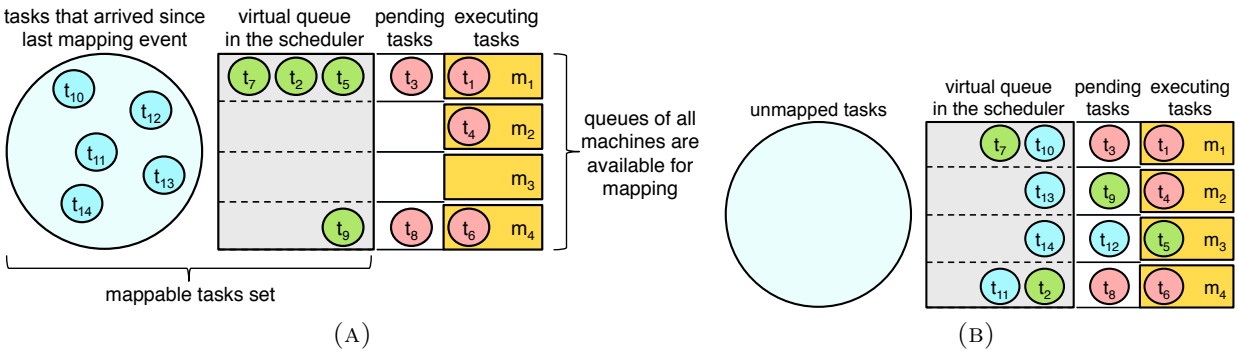


FIGURE 4.2. Example state of a four-machine system in the "queued" environment (a) before and (b) immediately after a mapping event.

machines as *available* choices when performing mapping decisions. Figure 4.2b shows an example state of the machines immediately after the mapping event is performed.

In the *polled* environment, individual machines do not have queues. Rather, at each mapping event, the machines are polled to check if they are currently idle or if they are executing a task. Only the machines that are idle are considered to be "available" for scheduling during the mapping event. The "mappable tasks set" in this environment comprises tasks that were either unmapped in the previous mapping event or newly arrived tasks since the last mapping event. In the polled environment, the only "unmovable tasks" are the tasks that are currently executing. Figure 4.3a shows an example state of a four-machine system in a polled environment prior to a mapping event, and Figure 4.3b shows a possible state of the system immediately after the mapping event.

4.3.2. DROPPING LOW UTILITY EARNING TASKS. We use a technique to drop tasks with low potential utility at the current time (introduced in our previous work [2]). *Dropping* a task means that it will never be mapped to a machine. Due to the oversubscribed environment, if a resource allocation heuristic tried to have all tasks execute, most of the task completion times would be so long that the utility of most tasks would decay significantly and be very small. This would negatively impact users as well as the overall system

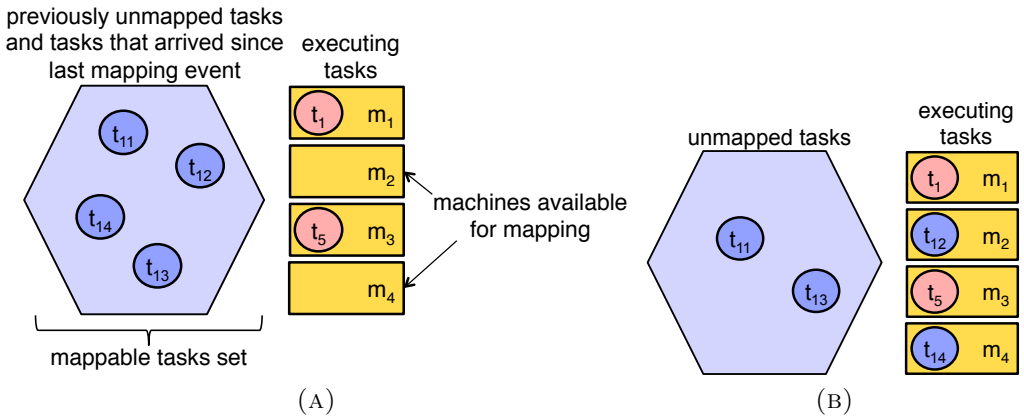


FIGURE 4.3. Example state of a four-machine system in the "polled" environment (a) before and (b) immediately after a mapping event.

performance. Given that the performance measure is the total utility achieved by summing the utilities of the completed tasks, dropping tasks leads to higher system performance, as well as more users that are satisfied.

The dropping operation determines the maximum possible utility that each mappable task could earn on any available machine assuming it can start as soon as possible, i.e., immediately after the unmovable tasks. If this utility is less than a *dropping threshold* (determined empirically), we drop this task from the set of mappable tasks. If the utility earned is not less than the threshold, the task remains in the mappable tasks set and is considered in the subsequent allocation decisions of the mapping event.

As our environment is oversubscribed, the number of tasks in the mappable tasks set increases quickly. With more tasks in the mappable set, the heuristics may take longer to perform their mapping decisions. This can delay the trigger of subsequent mapping events and result in poor performance because any newly arrived high utility tasks may not get serviced in a timely manner. Therefore, by dropping tasks with low potential utility, we reduce the size of the mappable tasks set and enable the heuristics to complete their

execution quicker and as a result trigger subsequent mapping events sooner. This allows the heuristics to promptly service any newly arriving high utility-earning tasks.

4.3.3. Energy Filtering. The goal of our energy filter technique is to remove potential allocation choices (task/machine/P-state combinations) from a heuristic's consideration if the allocation choice consumes more energy than an estimated fair-share energy budget (e_{bud}). The value of the e_{bud} needs to adapt based on the energy remaining in the day and the time remaining in the day. Therefore, the value of the e_{bud} is recomputed at the start of every mapping event.

We denote e_{cons} as the total energy that has been consumed by the system in the current day, and e_{unmov} as the energy that is guaranteed to be consumed, i.e., by unmovable tasks. The total energy that can be scheduled by heuristics (without violating the day's energy constraint) is denoted by e_{rem}. It is computed as $e_{rem} = E - (e_{cons} + e_{unmov})$.

To estimate e_{bud}, the filter also needs to compute the time remaining in the day within which the above energy can be consumed. The *availability time* of a machine is set to either the completion time of the last unmovable task on the machine or the current time, whichever is later. We compute the total time remaining for computations (τ_{rem}) by summing across machines the difference between the end time of the day and the availability time of the machine.

The average of the execution time values and energy values of all task types, machine types, and P-states is represented as $\bar{\tau}$ and $\bar{e}$, respectively. The energy filtering technique needs to estimate the total number of tasks that it can execute until the end of the day on average. Based on the time remaining, the estimated number of tasks that can complete on average was calculated as $\tau_{rem}/\bar{\tau}$ [36]. Similarly, based on the energy remaining, we now estimate the number of tasks that can complete on average as $e_{rem}/\bar{e}$, and use the minimum

of these two ratios as the number of tasks that the system can complete on average (n):

$$n = min\left(\frac{\tau_{rem}}{\bar{\tau}}, \frac{e_{rem}}{\bar{e}}\right). \tag{9}$$

To control the strictness of the filtering technique, we use a multiplier (λ). e_{bud} is computed using:

$$e_{bud} = \lambda \times \frac{e_{rem}}{n}. \tag{10}$$

When n is determined by the energy remaining in the system in Equation 9 ($e_{rem}/\bar{e}$), the second term of the product in Equation 10 appropriately reduces to using the average energy consumption of a task ($\bar{e}$) to determine e_{bud}.

Instead of using a fixed value for λ that needs to be empirically determined by running multiple parameter tuning experiments [36], we enable it to adapt based on the current rate of energy consumption and the target energy consumption rate. We denote the total compute time available in the system at the start of the day as T and its value is calculated by simply multiplying the number of machines by the total time in a day. The adaptive value for λ (which is recomputed at the start of every mapping event) is calculated as:

$$\lambda = \frac{E/T}{\left(\frac{e_{cons} + e_{unmov}}{T - \tau_{rem}}\right)}. \tag{11}$$

The goal of this adaptive parameter is to distribute the energy usage throughout the day. The numerator in Equation 11 is the target energy consumption rate and the denominator is the current average rate of energy consumption. If the current average rate is lower than the target rate, then that leads to a larger value for λ, which allows more energy to

be consumed by allocation choices. If the current average rate is higher than the target, the resulting smaller value of λ will only let low energy-consuming allocation choices pass through the filter. In this way, this adaptive technique automatically adjusts the level of filtering and can be deployed readily into any environment without the need for extensive off-line parameter tuning experiments.

4.3.4. Heuristics. We use the dropping operation with all the heuristics. We analyze the performance of the heuristics with and without the energy filtering technique. The heuristics are given the tasks and the task/machine/P-state choices that passed the dropping operation and the energy filtering technique (if used) to finally make assignments of the mappable tasks to the available machines. The *ready time* of a machine is the time by which it completes execution of the last task that is queued on it. In the polled environment, the ready time of the available machines is simply the current time. All of the heuristics progress iteratively, and in each iteration they make assignment for a mappable task to an available machine. The heuristics stop executing if either the set of mappable tasks is empty, or if there are no more available machines. In this work, we compare and analyze the performance of twelve heuristics. These heuristics assign tasks during mapping events while there still remains energy in the day. All heuristics ensure that the allocation they plan to make is a *valid assignment*, i.e., not assigning a task that cannot run on a particular special-purpose machine.

As most real-world schedulers assign tasks in an order based on their arrival time, we design and study the performance of a First Come First Served (*FCFS*) heuristic. The FCFS heuristic maintains a list of mappable tasks in an ascending order of arrival time. It then iteratively works through the list, each time making an assignment for the first task in the list. For the task being considered in each iteration, the heuristic assigns it to the available

machine that has the earliest ready time that is both a valid assignment, and that has passed the energy filter in the fastest P-state (i.e., P-state 0). In the polled environment, the heuristic picks a machine that is immediately available. The assigned task is removed from the sorted list and the next task is considered. We call this the ***FCFS P-state 0*** heuristic. We implemented another version of this heuristic that examines if the slower P-states pass through the energy filter in case the fastest P-state does not. We call this the ***FCFS All P-states*** heuristic.

The system examined in this study is oversubscribed, and the utility of tasks may start to decay once they arrive. Therefore, we consider an alternative to the FCFS heuristic that gives higher preference to the latest arrived task. This is the Last Come First Served ($LCFS$) heuristic. The LCFS heuristic is similar to the FCFS heuristic except that it maintains a list of mappable tasks in a descending order of their arrival times. We call the version that only permits the fastest P-state as the ***LCFS P-state 0*** heuristic and the version that allows any P-state to be chosen as the ***LCFS All P-states*** heuristic.

To account for the different importance levels of the tasks, we design prioritized versions of the FCFS and the LCFS heuristics. We envision that a version of these heuristics are implemented in real-world schedulers where the submitted jobs have different priority levels. The ***Prioritized-FCFS P-state 0*** heuristic first groups the mappable tasks based on their priority level (i.e., value of their initial maximum utility). It maintains a list of the tasks within each group in an ascending order of arrival time. The heuristic then considers the highest priority level group that contains any tasks. Considering those tasks in order, it makes assignment for each task to the earliest available machine that is a valid assignment and that has passed through the energy filter in P-state 0. After considering all the tasks in this group, it then considers the next highest priority level group that contains any tasks and

continually repeats this process. We designed another version of this heuristic that allows the other P-states to be considered as well. We call this the ***Prioritized-FCFS All P-states*** heuristic.

The prioritized versions of the LCFS heuristics are called the ***Prioritized-LCFS P-state 0*** and ***Prioritized-LCFS All P-states*** heuristics. These heuristics are similar to the Prioritized-FCFS heuristics with the difference that they maintain a list of tasks within each group in a descending order of their arrival times.

The ***Max Utility*** heuristic (designed based on the Min-Min technique [19, 54, 9]) gives preferences to mapping choices that earn the highest utility. The heuristic computes the utility that will be earned by each of the mappable tasks on each of the available machines in the different P-states. Each mappable task independently finds the machine/P-state choice that is valid, passes the energy filter, and that maximizes the utility earned. Among the different task/machine/P-state choices found, an assignment is made for the choice that earns the highest utility. The assigned task is removed from the set of mappable tasks and the process is repeated. This heuristic examines the utility function to find the utility that will be earned at the task completion time as opposed to the Prioritized heuristics that only look at the starting utility value of the task.

Unlike the Max Utility heuristic that solely maximizes for utility, the Max Utility-per-time (***Max UPT***) heuristic picks the task/machine/P-state choice that maximizes the ratio: "utility earned / execution time." In an oversubscribed environment, it is important to consider how much utility is earned per execution time used.

The Max Utility-per-Energy (***Max UPE***) heuristic focuses on reducing energy along with maximizing utility earned. It picks the allocation choice that maximizes the ratio: "utility earned / energy consumed by allocation." In an energy-constrained environment,

this heuristic helps to rank allocation choices by considering both the worth of a task and its energy consumption.

For comparison purposes, we implement a ***Random*** heuristic that randomly assigns a mappable task to an available machine in a random P-state (among the allocation choices that passed through the energy filter and are valid).

4.4. Simulation Setup

4.4.1. Overview. We use simulations to study our problem because we want to test and analyze the performance of the heuristics in a variety of environmental conditions. We simulate the arrival and mapping of tasks over a duration of 26 hours, with the first two hours used to bring the system up to steady-state operation. We collect our results (e.g., total utility earned, energy consumed) only from the start of the third hour to the end of the 26^{th} hour (total of 24 hours) to avoid the scenario where the machines start with empty queues. All the simulation experiments were run on the ISTeC Cray System at Colorado State University [46]. Each of the trials represents a new workload of tasks (with different utility functions, task types, and arrival times), and a different computing environment by using new values for the entries in the ETC and APC matrices (but without changing the number of machines). All of the parameters used in our simulations are set to closely match the expectations for future environments of interest to the ESSC.

4.4.2. Workload Generation. A utility function for each task in a workload is given, and each task has a maximum utility value (depending on its priority level) that starts at one of 8, 4, 2, or 1. These values are based on the plans of the ESSC, but for other environments, different number of values and different values of maximum utility may be used. A method

for generating utility functions can be found in [2]. Each task belongs to one among four urgency levels and 20 utility classes.

In our simulation environment, approximately 32,000 tasks arrive during the duration of a day, and each belongs to one of 100 task types. Out of the 100 task types, 83 are general-purpose and 17 are special-purpose. Each task type has approximately the same number of tasks in it. We generate the arrival patterns to closely match patterns of interest to ESSC [2]. The general-purpose tasks arrive in a sinusoidal pattern and special-purpose tasks follow a bursty arrival pattern.

4.4.3. Execution Time and Power Modeling. The compute system that we model has 13 machine types (four special-purpose) consisting of a total of 100 machines. The four special-purpose machine types have 2, 2, 3, and 3 machines in them. The remaining 90 machines are general-purpose and are split into the remaining nine machine types as follows: 5, 5, 5, 10, 10, 10, 10, 15, and 20. The machines of a special-purpose machine type run a subset of special-purpose task types approximately ten times faster on average than the general-purpose machines can run them (as discussed below). The special-purpose machines do not have the ability to run tasks of other task types. In our environment, three to five special-purpose task types are special for each special-purpose machine type.

We assume that heuristics can make use of three P-states in all machines: the highest power P-state (P-state 0), lowest power P-state, and an intermediate P-state. We use techniques from the Coefficient of Variation (COV) method [35] to generate the entries of the ETC and APC matrices in the highest power P-state. The mean value of execution time on the general-purpose and the special-purpose machine types is set to ten minutes and one minute, respectively. The mean dynamic power was set to 133 watts. To generate the dynamic power values for the intermediate P-state and the lowest power P-state, we

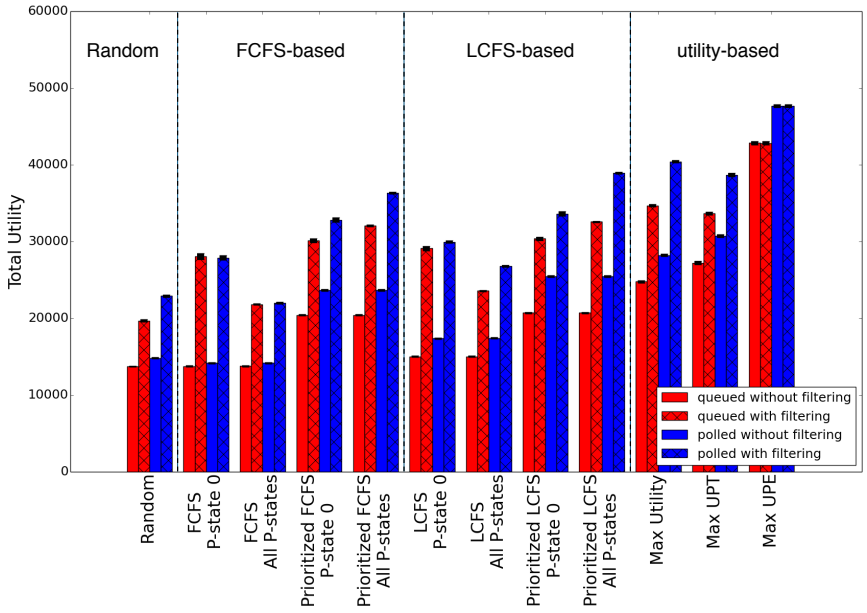


FIGURE 4.4. Total utility earned by the heuristics in the queued and the polled task management environments. For each of those cases, the hatched bars show the performance when the adaptive energy filter technique was used. For almost all the heuristics, the polled environment and the energy filtering technique help to significantly improve performance. The results are averaged over 48 simulation trials and 95% confidence intervals are computed.

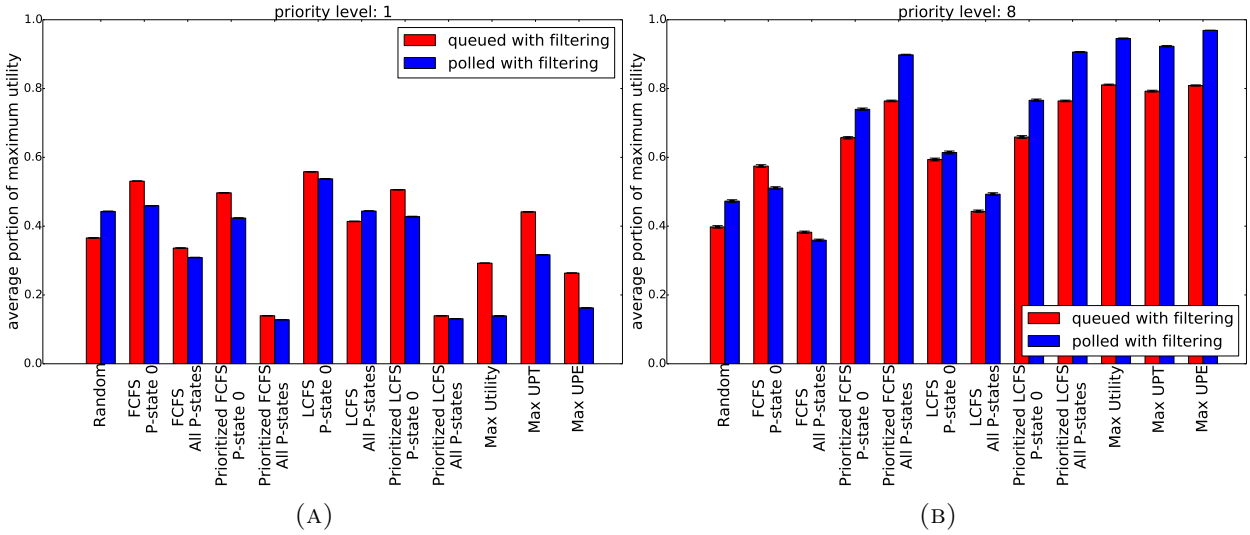
scale the dynamic power to 75% and 50%, respectively, of the highest power P-state. The execution times for these P-states are also generated by scaling the execution time at the highest power P-state by sampling a gamma distribution with a mean value that is approximately $1/\sqrt{(\%\ scaled\ in\ power)}$. For example, the lowest power P-state's execution time will be scaled by a value sampled from a gamma distribution that has a mean approximately equal to $1/\sqrt{0.5}$. The execution time of any task is guaranteed to be the shortest in the highest power P-state, but the most energy-efficient P-state can vary across tasks. Such a model approximates reality where the impact on execution time and energy consumption by switching P-states depends on, among other factors, the CPU-intensity/memory-intensity of the task and static power of the system.

4.4.4. Obtaining an Energy Constraint. For simulation purposes, we first run Max UPT (the heuristic that provides the best utility earned from our previous work [2]) for a full 24-hour time period, disregarding the energy constraint. We use 70% of the average of the total energy consumption throughout the day across 48 simulation trials as our energy constraint. As a result, for our simulations, we used an energy constraint value of 1.11 GJ per day.

4.5. Results

All results shown in this section display the average over 48 simulation trials with 95% confidence interval bars. The execution time of the heuristics and other mapping operations is on the order of 10^{-4} seconds per mapping event with the maximum time being $\approx$ 3 milliseconds. Therefore, in all of our cases, the mapping events were triggered approximately every 60 seconds. All of the heuristics used a dropping threshold of 0.5 units of utility to tolerate the oversubscription, as it gave the best performance without removing tasks of any one priority level completely. The dropping operation helped reduce the execution time of heuristics in the oversubscribed environment. When selecting a dropping threshold, one must consider the level of oversubscription of the environment in addition to the utility values of tasks.

Figure 4.4 shows the utility earned by the various heuristics in both queued and polled environments as well as with and without the energy filtering technique. The dashed vertical lines separate heuristic types into the following: Random, FCFS-based heuristics, LCFS-based heuristics, and the heuristics that use utility function information. We observe that, in general, the performance of the heuristics on the right is better than those on the left. The Prioritized FCFS heuristic performs better than the FCFS heuristic because it focuses



(A) (B)

FIGURE 4.5. Portion of maximum utility earned by the different heuristics when using the filtering technique for tasks that belong to (a) priority level 1, and (b) priority level 8.

on executing the high priority tasks first, and as a result, earns more utility per task. This is useful in an oversubscribed system as the heuristic is required to pick the better tasks to run. We see a similar trend when comparing Prioritized LCFS with LCFS. The LCFS-based heuristics usually outperform their FCFS counterparts because the LCFS-based heuristics focus on serving more recently arrived tasks. These recently arrived tasks tend to have utility values that have decayed less than tasks that arrived in the system earlier. The best performance is obtained by the heuristics that use utility function information to determine how much utility a task will earn. This is important because even though a task may have the highest priority it may also decay very fast. By the time such a task completes it may earn less utility than a task that may have a lower priority but does not decay much or does so slowly. The Max UPE heuristic significantly outperforms the other heuristics because it proactively reduces energy consumption and uses the saved energy to execute more tasks and earn more utility.

We observe that the polled environment significantly improves the performance for most of the heuristics, compared to the queued environment. This happens because the polled environment does not lock down a task into the pending slot, as happens in the queued environment, and therefore, is better able to more quickly serve any high utility earning tasks that arrive compared to the queued environment. It is worth noting that the polled environment has more idle time than the queued environment. This is because, in the polled environment, whenever a machine finishes execution of a task, it has to wait (idle) until the trigger of the next mapping event for another task to be assigned to it. Therefore, traditional metrics for performance such as utilization would incorrectly identify the polled environment to be performing worse, but it is those small amounts of idling distributed throughout the day that improves the ability of the system to quickly service any high utility tasks that may arrive. Figures 4.5a and 4.5b show on average the portion of maximum utility that was earned from tasks that belonged to priority level 1 and priority level 8, respectively. We see that for tasks that belong to the higher priority level, the polled environment does a better job of earning higher utility as compared to the queued environment. The figures also highlight the effectiveness of the priority-based and utility function-aware heuristics in attempting to earn a larger portion of the maximum utility from tasks that have a higher priority level versus those that have a priority level of 1. The trend with a priority level of 4 is very similar to 8 and the trend with a priority level of 2 is in-between the trends in the priority level 8 and 1 charts. The results shown in these figures are for the cases using the filtering technique but the no filtering cases also show similar overall trends.

It is worth noting that the time between subsequent mapping events versus the average task execution time is an important factor in determining which type of task management environment would be the best. For example, if the heuristic execution times were to be

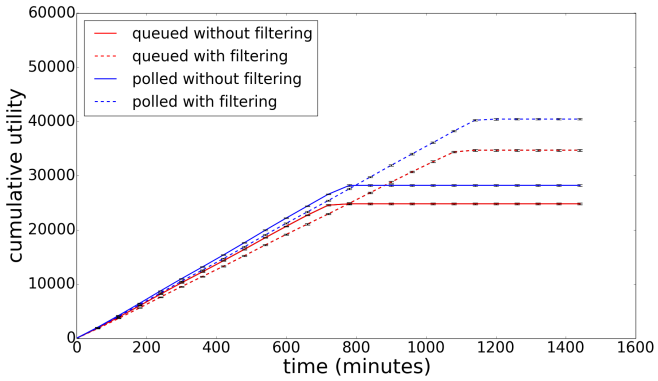


FIGURE 4.6. Cumulative utility earned by the Max Utility heuristic as time progresses highlighting the benefit of the filtering technique to save energy and use it to earn utility in the later parts of the day. The results are averaged over 48 simulation trials and 95% confidence intervals are computed.

very long, leading to longer times between mapping events, then it is likely that the queued environment will perform better than the polled environment because the polled environment will starve the machines whereas in the queued environment the machines can at least allocate tasks based on the queues.

Figure 4.4 also shows the significant benefit provided by our novel adaptive energy filtering technique for almost all of the heuristics in either of the task management environments. The filtering technique allows the heuristics to save energy for the later part of the day that can then be used to execute and earn utility from any high utility tasks that arrive at those times. A trace chart showing the utility earned by the Max Utility heuristic as time progresses (Figure 4.6) highlights the benefit of using the filtering technique. In our ESSC inspired environment, high-utility tasks arrive throughout the day and therefore it is typically beneficial to have some energy left to spend towards the end of the day. Similar trends are observed for the other heuristics as well and our filtering technique helps all of the heuristics in a similar manner by adapting to their energy consumption rate. When not using the filtering technique, all of the heuristics except Max UPE hit the energy constraint before the end of the day, and therefore, the heuristics (except Max UPE) benefit from using the

energy filter. As Max UPE already considers energy consumption, it does not benefit from the energy filter. Figure 4.6 shows how the polled environment always has a slightly higher slope than the queued environment. This is because the polled environment is consistently better able to serve the high priority tasks that arrive than the queued environment is able to serve.

For the FCFS-based and the LCFS-based heuristics, whether only P-state 0 (fastest P-state) is permitted or if all the other P-states are allowed only matters in the filtering cases, because in the no filtering case, the heuristic always makes assignments in P-state 0. When using filtering with the FCFS and LCFS heuristics, Figure 4.4 shows that considering only P-state 0 has a larger improvement in performance compared to the performance improvement when considering all P-states. This happens because of two reasons. The first reason is that by considering other P-states in all of the machines, even though more energy is saved, the execution time also increases leading to only a limited increase in total utility. It is worth noting that the first choice that satisfies the filter is chosen as opposed to the best energy choice. The second reason is that by only considering P-state 0, the FCFS and LCFS heuristics get the benefit of searching for another machine that satisfies the energy filter (but still only uses the fastest P-state). This search for a low energy machine also leads to minimizing the execution time in our environment (as energy = power $\times$ time). This results in more utility being earned from task completions. These benefits are not substantial when using the Prioritized FCFS and Prioritized LCFS heuristics, as they first consider the highest priority tasks that are not necessarily the earliest arriving or latest arriving, and the benefit from considering only P-state 0 is less than the benefit of the energy savings provided by considering other P-states, allowing them to earn more utility during the later part of the day.

4.6. Related Work

In [2], the concept of utility functions to describe a task's time-varying importance is introduced. Energy is not considered at all in that paper. In this work, we are concerned with maximizing utility while obeying an energy constraint.

Energy-aware scheduling has been extensively studied. In [55], the authors design techniques to schedule a bag-of-tasks to a heterogeneous computing system with the goal of minimizing energy consumption under a throughput constraint. In [50], the authors formulate a bi-objective resource allocation problem to analyze the trade-offs between makespan and energy consumption. Our work differs from these as we maximize utility earned under an energy constraint.

In [49], a set of dynamically arriving tasks with individual deadlines are allocated to machines within a cluster environment with the goal of conserving energy. Specifically, the authors try to optimize the energy consumption while meeting the constraint of completing all tasks by their deadlines. Our environment tries to maximize the total utility earned while operating under an energy constraint. As a result, we design an adaptive energy filtering technique. Additionally, [49] models an undersubscribed system, while our work focuses on highly oversubscribed environments.

The research in [56] attempts to maximize a mathematical model of Quality of Service under an energy constraint by using DVFS to take up slack time in an undersubscribed system, which is very different from our oversubscribed environment.

A dynamic resource allocation problem in a heterogeneous energy-constrained environment is studied in [10]. Tasks in this system contain individual deadlines, and the goal is to complete as many tasks by their individual deadlines as possible within an energy constraint. This is a different problem from our work as we are trying to maximize the utility earned

(based on each task's completion time) and not the number of tasks that meet their hard deadlines. The concept of an energy filter is used in [10], and we build on that for a more complex filter that automatically adjusts its level of filtering.

4.7. Conclusion

In this chapter, the problem of maximizing utility in an oversubscribed heterogeneous computing environment while satisfying an energy constraint was studied. The performance of multiple heuristics in different task management environments was examined and compared. The results indicate that the polled environment provides significant benefit over the queued environment in a system like this because it has the ability to quickly service newly-arrived tasks with high utility. The queued environment with its pending slot and virtual queues may be more useful in an environment where the time between mapping events is longer than the average task execution time. Multiple versions of the First Come First Served heuristic, that is commonly used in many real-world schedulers, were designed, implemented, and analyzed. The performance of these heuristics was compared with Last Come First Served heuristics and other smart heuristics and demonstrate the strength of these smart heuristics. A novel energy filtering technique was designed that can be used with any of the heuristics and can be readily deployed in any environment without the need for any off-line parameter tuning. The adaptive energy filtering technique improves the performance of almost all the heuristics by allowing the heuristics to distribute their consumption of the budgeted energy and earn more utility. Possible directions for future research are presented in Chapter 9.

CHAPTER 5

Bi-objective Optimization Between Utility and Energy Consumption[1]

5.1. Introduction

During the past decade, large datacenters (comprised of supercomputers, servers, clusters, farms, storage, etc.) have become increasingly powerful. As a result of this increased performance the amount of energy needed to operate these systems has also grown. It was estimated that between the years 2000 and 2006 the amount of energy consumed by high performance computing systems more than doubled [57]. In 2006 an estimated 61 billon kWh was consumed by servers and datacenters, approximately equal to 1.5% of the total United States energy consumption for that year. This amounted to $4.5 billion in electricity costs [57]. Since 2005, the total amount of electricity used by HPC systems increased by another 56% worldwide and 36% in the U.S. Additionally, during 2010, global HPC systems accounted for 1.5% of total electricity use, while in the U.S., HPC systems accounted for 2.2% [58].

With the cost of energy and the need for greater performance rising, it is becoming increasingly important for HPC systems to operate in an energy-efficient manner. One way to reduce the cost of energy is to minimize the amount of energy consumed by a specific system. In this work, it is shown that one can reduce the amount of energy consumed by

[1]This work was done jointly with Ph.D. student Bhavesh Khemka. The full list of co-authors is at [45]. This work was supported by Oak Ridge National Laboratory and their Extreme Scale Systems Center under subcontract numbers 4000094858 and 4000108022, and by a National Science Foundation Graduate Research Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research also used resources of the Extreme Scale Systems Center at Oak Ridge National Laboratory as well as the CSU ISTeC Cray System supported by NSF Grant CNS-0923386.

a system by making intelligent scheduling decisions. Unfortunately, consuming less energy often leads to a decrease in the performance of the system [59]. Thus, it can be useful to examine the trade-offs between minimizing energy consumption and maximizing computing performance for different resource allocations. Current resource managers, such as MOAB, cannot reasonably determine the trade-offs between performance and energy.

In this research, a computing environment and corresponding workload that is being investigated by the Extreme Scale Systems Center (ESSC) at Oak Ridge National Laboratory (ORNL) is modeled. The ESSC is a joint venture between the United States Department of Defense (DoD) and Department of Energy (DOE) to provide research, tools, software, and technologies that can be utilitized in both DoD and DOE environments. The goal is to design an analysis framework that a system administrator can use to analyze the energy and performance trade-offs of a system by using different resource allocations (i.e. mapping of tasks to machines). System performance is measured in terms of total utility earned, where each task in the workload is assigned a time utility function that monotonically decreases in value the longer a task remains in the system [60]. The computing environment is a heterogeneous mix of machines and tasks that represents an environment with system characteristics and workload parameters that are based on the expectations of future DoD and DOE environments.

In a heterogeneous environment, tasks may have different execution and power consumption characteristics when executed on different machines. One can change the performance and energy consumption of the system by using different resource allocations. A resource allocation is defined to be a complete mapping of tasks to machines, where it is assumed the number of tasks is much greater than the number of machines. To create these resource allocations, this scheduling problem is modeled as a bi-objective optimization problem that

maximizes utility earned and minimizes energy consumed. The Nondominated Sorting Genetic Algorithm II (chap5nsga-II) [61] has been adapted to create the resource allocations.

To analyze the effect of different resource allocations, this method has been implemented in a static and offline environment. To create the offline environment a trace over a specified time period of the modeled environment is simulated. This allows information to be gathered about the number of tasks that arrived during this time period, as well as the arrival times and types of tasks that were present in the system. The availability of this information makes this a static resource allocation problem. The knowledge gained from studies such as this can be used to set the parameters needed for designing dynamic or online allocation heuristics.

Execution and power consumption characteristics from real machines and applications are utilized in this study. This real characteristic data is used to create a larger data set in order to simulate larger systems. In all cases, the results show that by using different resource allocations a system administrator can greatly change the amount of utility earned and power consumed based on the needs of their system.

In this chapter the following contributions are made:(a) modeling a bi-objective resource allocation problem between total utility earned and total energy consumed to address concerns about energy-efficient computing, specifically for environments of DoD and DOE interest, (b) creating and evaluating many intelligent resource allocations to show that the utility earned and energy consumed by the system can change greatly, (c) demonstrating the method by using real machine and task data, (d) providing a method to create synthetic data sets that preserve the heterogeneity measures found from real data sets, and (e) analyzing the effect of using different seeding heuristics on the evolution of solutions found by the genetic algorithm.

The remainder of this chapter is organized as follows. Related work is discussed in Section 5.2. The system model and real data set are explained in Section 5.3. In Section 5.4, the bi-objective optimization problem and the NSGA-II are described. The simulation setup is detailed in Section 5.5. Section 5.6 analyzes the simulation results. Finally, the conclusion will be given in Section 5.7.

5.2. Related Work

The authors of [59] formulate a bi-objective resource allocation problem to analyze the trade-offs between makespan and energy consumption. Their approach is concerned with minimizing makespan as their measure of system performance, opposed to maximizing utility in our approach. Additionally, they model an environment where the workload is a bag of tasks, not a trace from a dynamic system. This is important because they do not consider arrival times or the specific ordering of tasks on machine. Finally, the authors do not demonstrate their approach using real historical data.

A bi-objective optimization problem between makespan and reliability is used to solve heterogeneous task scheduling problems in [62] and [63]. We perform the bi-objective optimization between utility earned and energy consumed.

The study in [64] implements a weighted sum simulated annealing heuristic to solve a bi-objective optimization problem between makespan and robustness. One run of this heuristic produces a single solution, and different weights can be used to produce different solutions. This differs from our approach in that we independently evaluate our two objective functions that allows us to create a Pareto front containing multiple solutions with one run of our algorithm.

In [65], the authors minimize makespan and total tardiness to solve the bi-objective flowshop scheduling problem. The authors utilize a Pareto-ant colony optimization approach to create their solutions. This work differs from ours by not using a genetic algorithm nor is it concerned with utility nor energy consumption.

In [66], a job-shop scheduling problem is modeled as a bi-objective optimization problem between makespan and energy consumption. The authors model a homogeneous set of machines whereas our work models a heterogeneous set of machines. The work in [66] also differs from ours by using an algorithm that only produces a single solution.

Heterogeneous task scheduling in an energy-constrained computing environment is examined in [8]. The authors model an environment where devices in an ad-hoc wireless network are limited by battery capacity. The heuristics in [8] create a single solution while ours creates multiple solutions. In our study, we are not concerned with an energy-constrained system, but instead we try to minimize the total energy consumed.

Minimizing energy while meeting a makespan robustness constraint in a static resource allocation environment is studied in [67]. This work is not an explicit bi-objective optimization. Our work differs by using utility maximization as an objective instead of minimizing makespan.

A dynamic resource allocation problem in an energy-constrained environment is studied in [10]. Solutions to this problem must complete as many tasks as they can while staying within the energy budget of the system. In our work, we model a trace of a dynamic system allowing us to create a static allocation. We also do not constrain the amount of energy our system can consume.

Mapping tasks to computing resources is also an issue in hardware/software co-design [68]. This problem domain differs from ours, however, because it typically considers the

hardware design of a single chip. Our work assumes a given collection of heterogeneous machines.

5.3. System Model

5.3.1. Overview. Our system environment is modeled based on the needs of the ESSC at ORNL. The system is intended to provide a model of both the machines and workload used in such an environment. This system model has been designed with detailed collaboration between members of Colorado State University, ORNL, and the United States DoD to ensure it accurately captures the needs and characteristics of the ESSC.

5.3.2. Machines. This model consists of a suite of M heterogeneous machines, where each machine in this set belongs to a specific machine type μ. Machine types exhibit heterogeneous performance, that is machine type A may be faster than machine type B for some task types but slower for others [35]. Machines types also exhibit heterogeneous energy consumption and belong to one of two categories. The first category is general-purpose machines. General-purpose machines are machines that are able to execute any of the task types in the system, and they make up the majority of the machines in the environment. The other category of machines is special-purpose machines. Machines within this category can only execute a small subset of task types and are incapable of executing the remaining task types. Special-purpose machines generally exhibit a 10x decrease in the execution times of the task types they can execute compared to the general-purpose machines. The heterogeneity between the machine types can be attributed to differences in micro-architectures, memory modules, hard disks, and/or other system components.

5.3.3. Workload. In the ESSC environment, tasks arrive dynamically throughout the day. Once a task arrives, the utility earned by a task may start to decay, see Subsection 5.4.2.1. Utility dictates how much useful work a given task can accomplish. Utility is represented by a monotonically-decreasing function with respect to time. Therefore the sooner a task completes execution the higher utility it might earn [60]. Because we are performing a bi-objective analysis of the system, we consider a trace of tasks that arrive into the system within a specified amount of time (e.g., one hour). The arrival times of each task in the trace must be recorded to accurately calculate how much utility a given task earns.

Every task in the trace is a member of a given task type. Each task type has unique performance and energy consumption characteristics for executing on the machine types. Similar to the machine types, task types belong to one of two categories; general-purpose tasks types and special-purpose tasks types. General-purpose tasks types are task types that can only execute on the general-purpose machine types. A special-purpose task type can execute on a specific special-purpose machine type at an increased rate of execution (compared to the general-purpose machine type), and it is also able to execute on the general-purpose machine types.

5.3.4. Execution and Energy Consumption Characteristics. It is common in resource allocation research to assume the availability of information about the performance characteristics of the tasks types and machine types present in a system (e.g., [12, 15, 13, 14]). This information is contained within an Estimated Time to Compute (ETC) matrix. An entry in this matrix, ETC(τ,μ), represents the estimated time a task of type τ will take to execute on a machine of type μ. The values contained within this matrix can be synthetically created to model various heterogeneous systems [35], or the values can be obtained from

historical data (e.g., [13, 15]). The assumption that ETC values can be obtained from historical data is valid for the intended ESSC environments.

In this study, we also require information about the power consumption characteristics of the task types and machine types. We call this set of data the Estimated Power Consumption (EPC) matrix. An entry in this matrix, EPC(τ,μ), represents the average amount of power a task of type τ consumes when executing on a machine of type μ. Different EPC values can represent different task type energy characteristics, e.g., computationally intensive tasks, memory intensive tasks, or I/O intensive tasks. The values within the EPC matrix can also be created synthetically or gathered from historical data. In this study we utilize ETC and EPC matrices that contain both real historical data (described in Section 5.3.4.1), and synthetic data derived from the historical data (described in Section 5.3.4.2).

5.3.4.1. *Gathering of Historical Data.* To accurately model the relationships between machine performance and energy consumption in a heterogeneous suite of machines, we first create ETC and EPC matrices filled with historical data. For this data, we use a set of online benchmarks [69] that tested a suite of nine machines (Table 5.1) over a set of five tasks (Table 5.2). The machines differ by the CPU and motherboard/chipset used, but all the machines use the same amount of memory (16GB) and the same type of hard drive and GPU. For each task, the benchmark produced the average execution time for that task on each of the machines as well as the average power consumed by that task on each of the machines. We were able to place these values into our ETC and EPC matrices. Each of the nine machines from the benchmark represents a specific machine type, and each of the five tasks from the benchmark represents a specific task type. This data provides us with initial ETC and EPC matrices of size 5×9.

TABLE 5.1. Machines (designated by CPU) used in benchmark

AMD A8-3870k
AMD FX-8159
Intel Core i3 2120
Intel Core i5 2400S
Intel Core i5 2500K
Intel Core i7 3960X
Intel Core i7 3960X @ 4.2 GHz
Intel Core i7 3770K
Intel Core i7 3770K @ 4.3 GHz

TABLE 5.2. Programs used in benchmark

C-Ray
7-Zip Compression
Warsow
Unigine Heaven
Timed Linux Kernel Compilation

5.3.4.2. *Creation of Synthetic Data.* From the historical data, we are able to derive a larger data set to study how our algorithm performs for a more complex problem. To create a larger data set, we need to increase the number of task types as well as introduce special-purpose machines. The real historical data will be included in this larger data set. For the new synthetic data set to resemble the real historical data as closely as possible, we need to preserve the heterogeneity characteristics of the historical data set. Heterogeneity characteristics of a data set can be measured using various standard statistical measures, such as the coefficient of variation, skewness, and kurtosis [70]. Two data sets that have similar heterogeneity characteristics would have similar values for these three measures.

To create a larger data set (we describe the process using the ETC matrix, but the process is identical for the EPC matrix), our first task is to create more task types. To do this, we first calculate the average execution time of each of the real task types across all the machines, this is also known as the row average of a task type. We then use these row average task execution times as the basis for creating new task types. We calculate the following

heterogeneity measures: mean, variation, skewness, and kurtosis (mksv) for the collection of row average task execution times. With the mvsk values we use the Gram-Charlier expansion [71] to create a probability density function (PDF) that produces samples of row average task execution times. By sampling this PDF we can create the row average task execution times for any number of new task types.

Once a desired number of task types are created, the next step is to populate the ETC entries for these new task types. While doing this, we want to preserve the relative performance from one machine to another. We calculate the task type execution time ratio. This ratio is the execution time of a specific task type on a specific machine, divided by the average task execution time (across all machines) for that task type. For example, let us assume task type 1 takes eight minutes to execute on machine type A, but it takes twelve minutes to execute on machine type B. Also assume task type 1 has an average execution time of ten minutes across all machines. On machine type A the task type has a task type execution time ratio of .8 while on machine type B, the task type has a task type execution time ratio 1.2. We calculate this ratio for all of the real task types on all of the real machines. Typically, faster machines will have values less than one for this ratio while slower machines have values greater than one.

Having found the relative performance of the machines to one another, the next step is to capture the task heterogeneities across the individual machines to create the new ETC matrix values. The following procedure is performed individually for each machine. On a given machine, we calculate the heterogeneity values of the task type execution time ratios for the real task types. By using the mvsk values produced, we then create a PDF that produces samples of task type execution time ratios for that specific machine. We sample this PDF to create task type execution time ratios for the new task types on that specific

machine. We can then take the task type execution time ratios for the new task types and multiply them by their respective average task type execution time values to produce the actual ETC values for the new task types.

The final step is to now create the special-purpose machines types. Based on the expectations of future DoD and DOE environments, special-purpose machine types are modeled to perform around 10x faster than the general-purpose machine types for a small number of task types (two to three for each special purpose machine type). To create these machines we first select the number of task types that can be executed on the special-purpose machines types. Then the average execution time (across all the machines) for each of these selected task types is found. The average execution time for each task type is divided by ten and is then used as the ETC value for the special-purpose machine type. When calculating EPC values, the average power consumption across the machines is *not* divided by ten.

This method allows us to create larger data sets that exhibit similar heterogeneity characteristics when compared to the real data. This method can be used to create both ETC and EPC matrices, with the exception of special-purpose EPC values as stated above.

5.4. Bi-Objective Optimization

5.4.1. Overview. In many real world problems, it is important to consider more than one goal or objective. It is often the case that these multiple objectives can directly or indirectly compete with each other. Optimizing for one objective may cause the other objective to be negatively impacted. In this work we try to maximize utility earned while minimizing energy consumed. We will show in our results that these two objectives do compete with each other. In general, a well-structured resource allocation that uses more energy will earn more utility and one that uses less energy will earn less utility. This relationship occurs

because we model utility as a monotonically-decreasing performance measure, that is, the longer a task takes to execute, the less utility it may earn. In [59], it is shown that spending more energy may allow a system to complete all the tasks within a batch sooner.

In Section 5.4.2, we define the two objective functions we are optimizing. Section 5.4.3 describes which solutions should be considered when solving a bi-objective optimization. The genetic algorithm used for this study is briefly explained in Section 5.4.4.

5.4.2. Objective Functions.

5.4.2.1. *Maximizing Utility.* One objective is maximizing total system utility earned. Total Utility Earned is a metric used to measure the performance of a computing system. Utility can be thought of as the amount of useful work a system accomplishes [60]. The amount of utility a task earns is determined by the completion time of the task, as well as by three sets of parameters; priority, urgency, and utility characteristic class. These parameters form what is called a time-utility function. This function is a monotonically-decreasing function [60].

Priority represents the total amount of utility a task could possibly earn, i.e., how important a task is. Urgency represents the rate of decay of utility a task may earn as a function of completion time. Utility characteristic class allows the utility function to be separated into discrete intervals. Each interval can have a beginning and ending percentage of maximum priority, as well as an urgency modifier to control the rate of decay of utility. The definition of these parameters are based on the needs of the ESSC. The value of these parameters in an actual system are determined by system administrators (not individual users) and are policy decisions that can be adjusted as needed.

Figure 5.1 illustrates a sample task time-utility function. We see that the function is monotonically decreasing. Additionally, we see the different intervals existing in the function.

By evaluating the function at different completion times, we can determine the amount of utility earned. For example, if a task finished at time 20, it would earn twelve units of utility, whereas if the task finished at time 47, it would only earn seven units of utility.

Every task t in the system is assigned its own utility function $\Upsilon(t)$. This function returns the utility earned by that task when it completes execution. Tasks that have hard deadlines can be modeled as having their utility decay to zero by a specific time. To optimize for total utility, the goal is to maximize the sum of utilities earned over all tasks in the system. Given that there are T tasks in the system, total utility earned, denoted U, can be defined as

$$U = \sum_{\forall t \in T} \Upsilon(t). \tag{12}$$

5.4.2.2. *Minimizing Energy Consumed.* The other objective we optimize for is minimizing total energy consumed. For a given resource allocation, the total energy consumed is the sum of the energy consumed by each task to finish executing. We first calculate the Expected Energy Consumption (EEC) for a given task on a given machine. This is found by multiplying the ETC and EPC values for that task on that machine. Assume the function $\Phi(t)$ returns the task type of a given task t and the function $\Omega(m)$ returns the machine type of a given machine. The expected energy consumption of task t on machine m can then be given as

$$EEC[\Phi(t), \Omega(m)] = ETC[\Phi(t), \Omega(m)] \times EPC[\Phi(t), \Omega(m)]. \tag{13}$$

Let T_m be the tasks that execute on machine m, where task $t_m \in T_m$. The total energy consumed by the system, denoted E, is found by

$$E = \sum_{\forall m \in M} \sum_{\forall t_m \in T_m} EEC[\Phi(t_m), \Omega(m)]. \tag{14}$$

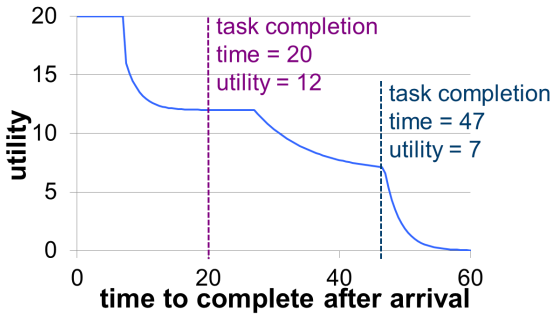


FIGURE 5.1. Task time-utility function showing values earned at different completion times.

5.4.3. GENERATING SOLUTIONS FOR A BI-OBJECTIVE OPTIMIZATION PROBLEM. When a problem contains multiple objectives, it is challenging to determine a single global optimal solution. Often, there is instead a *set* of optimal solutions. This set of optimal solutions may not be known, thus it is important to find a set of solutions that are as close as possible to the optimal set. The set of solutions that we find that best approximates the optimal set is called the set of Pareto optimal solutions [72]. The Pareto optimal set can be used to construct a Pareto front that can illustrate the trade-offs between the two objectives.

Not all solutions we find can exist within the Pareto optimal set. We select the eligible solutions by examining solution dominance. Dominance is used to compare two solutions to one another. For one solution to dominate another, it must be better than the other solution in at least one objective, and better than or equal in the other objective. Figure 5.2 illustrates the notion of solution dominance. In this figure we show three potential solutions: A, B, and C. The objectives we are optimizing for are minimizing total energy consumption along the x-axis and maximizing total utility earned along the y axis. A "good" solution would be one that consumes small amounts of energy and earns large amounts of utility (upper left corner).

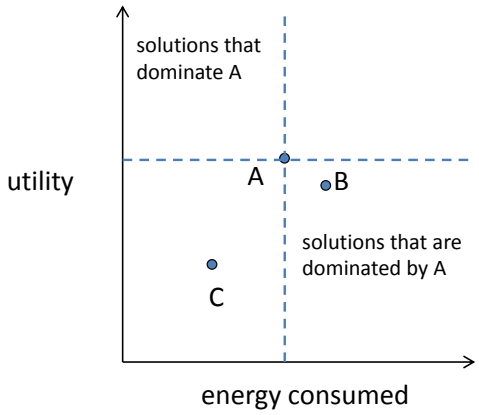


FIGURE 5.2. Illustration of solution dominance for three solutions: A, B, and C. Solution A dominates solution B because A has lower energy consumption as well as it earns more utility. Neither solution A nor C dominate each other because C uses less energy, while A earns more utility.

If we consider solutions A and B, we can say that B is dominated by A. This is because A uses less energy as well as earns more utility than B. The same is true for any solution located within the lower right region. Now, when we examine solutions A and C we cannot make any claims on the dominance of one solution over the other. This is because C uses less energy than A, but C does not earn as much utility, thus the two solutions can not be directly compared to each other, which means they are both located on the Pareto front. Finally, solution A will only be dominated by solutions that exist in the upper left region.

5.4.4. NONDOMINATED SORTING GENETIC ALGORITHM II. The Nondominated Sorting Genetic Algorithm II (chap5nsga-II) was designed for solving bi-objective optimization problems [61]. We adapt a version of the NSGA-II that we used in [59] for a different, simpler resource allocation problem. We will briefly describe the basic NSGA-II algorithm and how we have modified it for our specific problem.

The NSGA-II is a popular multi-objective genetic algorithm that utilizes solution dominance to allow the populations of chromosomes to evolve into better solutions over time.

For our problem domain, a single population represents a set of possible resource allocations (solutions). To use the NSGA-II for our problem we needed improve upon [59] and encode the algorithm so that it could solve bi-objective resource allocation problems where the two objectives are maximizing utility earned and minimizing energy consumed. To do this, the algorithm must be able to accurately track and calculate the individual utility earned by each task. This meant we needed to create our own genes, chromosomes, crossover operator, and mutation operator.

Genes represent the basic data structure of the genetic algorithm. For our problem, a gene represents a task. Each gene contains: the machine the gene will operate on, the arrival time of the task, and the global scheduling order of the task. The global scheduling order is a number from 1 to the number of tasks in the chromosome and it controls the order that tasks execute on the individual machines. The lower the number, the sooner the task will execute. The global scheduling order is independent of the task arrival times.

A chromosome represents a complete solution, i.e., resource allocation. Each chromosome is comprised of T genes, where T is the number of tasks that are present in the trace being studied. The i^{th} gene in every chromosome corresponds to the same task, in particular, the i^{th} task ordered based on task arrival times. To examine dominance relationships amongst the chromosomes in the population, each chromosome is individually evaluated with respect to utility earned and energy consumed. Because the global scheduling order is independent of task arrival times, we must ensure that any task's start time is greater than or equal to its arrival time. If this is not the case, the machine sits idle until this condition is met.

For populations and chromosomes to evolve from one generation to the next, the following crossover and mutation operations were implemented. For crossover, we first select two chromosomes uniformly at random from the population. Next, the indices of two genes

are selected uniformly at random from within the chromosomes. Finally, we swap all the genes between these two indices, from one chromosome to the other. In this operation, the machines the tasks execute on, and the global scheduling orders of the tasks are all swapped. The goal of this operation is to allow chromosomes making good scheduling decisions to pass on favorable traits to other chromosomes.

For the mutation operation, we randomly select a chromosome from the population. We then select a random gene from within that chromosome. We then mutate the gene by selecting a random machine that that task can execute on. Additionally, we select another random gene within the chromosomes and then swap the global scheduling order between the two genes.

Within a given population, a solution is ranked based on how many other solutions dominate it. If no other solutions dominate a solution, it is said to be nondominated and is given a rank of 1. A solution's rank can be found by taking *1 + the number of solutions that dominate it.* All solutions of rank 1 within a given population represent the current Pareto optimal set. The process of sorting the solutions is performed using the nondominating sorting algorithm, and is one step of the NSGA-II algorithm. A basic outline of the NSGA-II is taken from [59] and shown in Algorithm 1.

An important step to understand in the algorithm is the creation of offspring populations (step 3). We start with a parent population of size N. We then perform $N/2$ crossover operations (each crossover operation produces two offspring) to produce an initial offspring population also of size N. Next, the mutation operation is then performed with a probability (selected by experimentation) on each offspring within the population. If a mutation occurs, only the mutated offspring remains in the population.

Algorithm 1 chap5nsga-II algorithm from [59]

1: create initial population of N chromosomes
2: **while** termination criterion is not met **do**
3: create offspring population of size N
4: perform crossover operation
5: perform mutation operation
6: combine offspring and parent populations into a single meta-population of size $2N$
7: sort solutions in meta-population using nondominated sorting algorithm
8: take all of the rank 1, rank 2, etc. solutions until we have at least N solutions to be used in the parent population for the next generation
9: **if** more than N solutions **then**
10: for solutions from the highest rank number used, take a subset based on crowding distance [61]
11: **end if**
12: **end while**
13: the final population is the Pareto front used to show the trade-offs between the two objectives

We can now combine the parent and offspring population into a single meta-population of size $2N$ (step 6). This combining of populations allows elitism to be present within the algorithm, that is the algorithm keeps the best chromosomes from the previous generation and allows them to be evaluated in the current generation. From this new meta-population, we then select the next generation's parent population (steps 7, 8, 9, and 10). To create the new parent population, we need to select the best N chromosomes from the meta-population.

To illustrate this procedure, we provide the following example. Assume we currently have a parent population and spring population, each with 100 chromosomes, which combines to make a meta-population with 200 chromosomes. We then perform the nondominated sorting algorithm from step 7. We find that we have 60 chromosomes of rank 1, 30 chromosomes of rank 2, 20 chromosomes of rank 3, and 90 chromosomes that have a rank higher than 3. We need to create a new parent population with only 100 chromosomes in it. First we take the 60 chromosomes that are rank 1 and place them into the new population. This leaves room for 40 more chromosomes. We then place the rank 2 chromosomes in the new population.

There is space in the population for 10 additional chromosomes. To select only 10 out of the 20 rank 3 chromosomes we use the method of crowding distance [61] to arrive at our full 100 chromosome population. Crowding distance is a metric that penalizes chromosomes that are densely packed together, and rewards chromosomes that are in remote sections of the solutions space. This operation creates a more equally spaced Pareto front.

5.5. Simulation Setup

5.5.1. Datasets and Experiments. To illustrate how our analysis framework allows system administrators to analyze the trade-offs between utility earned and energy consumed of a given system, we have conducted numerous simulations using three different sets of data.

The first set consists of the real historical ETC and EPC data gathered from the online benchmarks (nine machine types and five task types) [69]. This set only allotted one machine to each machine type and simulated 250 tasks comprised of the five task types arriving over a period of 15 minutes. We are performing a post-mortem static resource allocation as we are using a trace of a (simulated) system, thus the arrival times of all tasks are known *a priori*. This real data set is used as the basis for the second and third data sets, as described is Subsection 5.3.4.2.

The second and third sets consist of the ETC and EPC data manufactured from the real data (Subsection 5.3.4.2). For these sets, we created four special-purpose machine types for a total of 13 machine types and 25 additional task types for a total of 30 task types. Additionally, for both sets there were 30 machines across the 13 machine types. The break up of those machines can be seen in Table 5.3. Data sets 2 and 3 differ from one another by the number of tasks each set simulates. Set 2 simulates 1000 tasks arriving within the span of 15 minutes, while set 3 simulates 4000 tasks arriving within the span of an hour.

TABLE 5.3. Breakup of machines to machine types

machine type	number of machines
Special-purpose machine A	1
Special-purpose machine B	1
Special-purpose machine C	1
Special-purpose machine D	1
AMD A8-3870k	2
AMD FX-8159	3
Intel Core i3 2120	3
Intel Core i5 2400S	3
Intel Core i5 2500K	2
Intel Core i7 3960X	4
Intel Core i7 3960X @ 4.2 GHz	2
Intel Core i7 3770K	5
Intel Core i7 3770K @ 4.3 GHz	2

We conducted experiments on each data set. For the first group of experiments, we let the genetic algorithm execute for a given number of generations and then we analyze the trade-offs between utility earned and energy consumed. The second group of experiments compare the effect of using different seeds within the initial population on the evolution of the Pareto fronts. The seeding heuristics used for these experiments are described in the section below.

5.5.2. SEEDING HEURISTICS. Seeding heuristics provide the genetic algorithm with initial solutions that try to intelligently optimize one or both of the objectives. The seeds may help guide the genetic algorithm into better portions of the search space faster than an all random initial population. We implement the following four heuristics: Min Energy, Max Utility, Max Utility-per-Energy, and Min-Min Completion Time. The execution times of the greedy heuristics are negligible compared to the NSGA-II, making solutions found by these heuristics plausible to use. To use a seed within a population, we generate a new chromosome from one of the following heuristics. We place this chromosome into the population and create the rest of the chromosomes for that population randomly.

5.5.2.1. *Min Energy.* Min Energy is a single stage greedy heuristic that maps tasks to machines that minimize energy consumption. This heuristic maps tasks according to their arrival time. For each task the heuristic maps it to the machine that consumes the least amount of energy to execute the task. This heuristic will create a solution with the minimum possible energy consumption. Solutions may exist that consume the same amount of energy while earning more utility.

5.5.2.2. *Max Utility.* Max Utility is also a single stage greedy heuristic similar to the min energy heuristic except that it maps tasks to the machines that maximizes utility earned [60]. This heuristic must consider the completion time of the machine queues when making mapping decisions. There is no guarantee this heuristic will create a solution with the maximum obtainable utility.

5.5.2.3. *Max Utility-per-Energy.* Max Utility-per-Energy tries to combine aspects of the previous two heuristics. Instead of making mapping decisions based on either energy consumption or utility earned independently, this heuristic maps a given task to the machine that will provide the most utility earned per unit of energy consumed.

5.5.2.4. *Min-Min Completion Time.* Min-min completion time is a two-stage greedy heuristic that maps tasks to the machines that provide the minimum completion time [19, 54, 9]. During the first stage, the heuristic finds for every task the machine that minimizes that task's completion time. In the second stage, the heuristic selects from among all the task-machine pairs (from the first stage) the pair that provides the overall minimum completion time. That task is then mapped to that machine, and the heuristic repeats this operation until there are no more tasks to map.

5.6. Results

In Figures 5.3, 5.4, and 5.6 we show the location of numerous Pareto fronts. In each figure we show different fronts corresponding to different initial populations, for each of our three data sets respectively. Each of the four subplots show the Pareto fronts through a specific number of NSGA-II iterations. The x-axis is the number of megajoules consumed by the system, and the y-axis is the amount of utility earned by the system. Each marker within the subplots represents a complete resource allocation. Each marker style represents a different population. The diamond marker represents the population that contained the "Min Energy" seed, the square marker represents the population with the "Min-Min completion time" seed, the circle marker represents the population with the "Max Utility" seed, the triangle maker represents the population with the "Max Utility-per-Energy" seed, and finally the star marker represents the population with a completely random initial population. We also considered an initial population that contained all four of the seeding heuristics, but we found that this population performed similarly to the min-energy seeded population, and thus did not include it in our results.

Figure 5.3 shows the Pareto fronts for the real historical data set. For this data set we evaluated the evolution of the Pareto fronts through four different number of iterations; 100, 1000, 10,000, and 100,000 iterations. First, in the top left subplot we see that after 100 iterations the populations have formed distinct Pareto fronts covering various parts of the solution space. This occurs because of the use of the different seeds within each population. The presence of the seed within a population allows that population to initially explore the solution space close to where the seed originated. As the number of iterations increase though, the presence of the seed starts to become irrelevant because all the populations, even the all random initial population, start converging to very similar Pareto fronts.

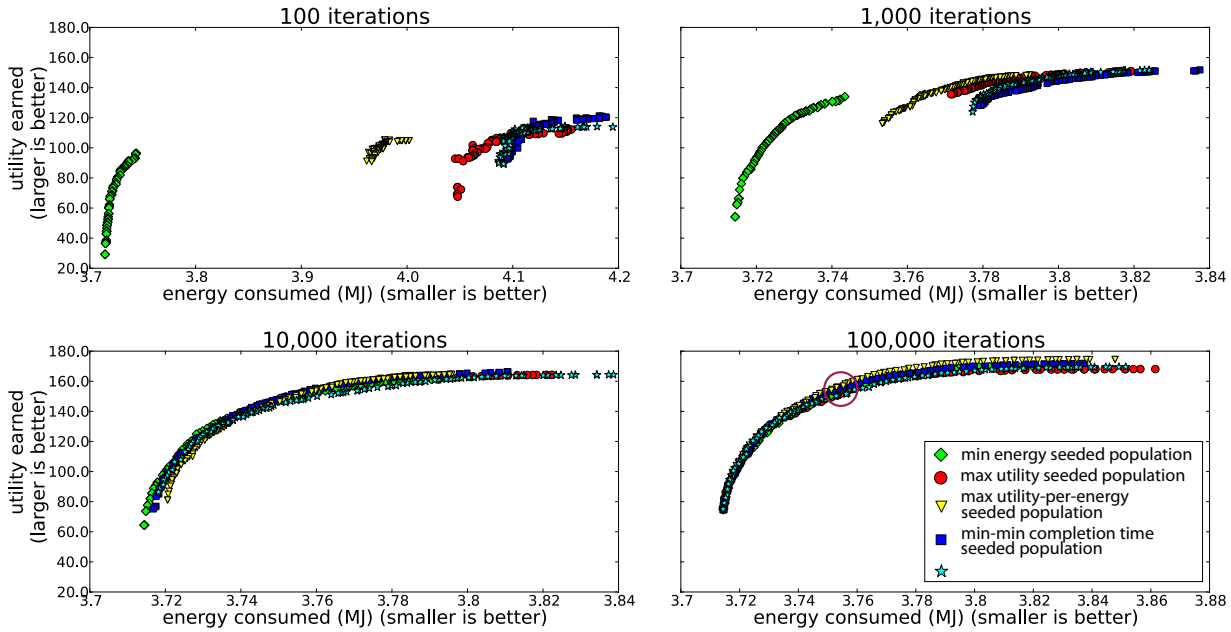


FIGURE 5.3. Pareto fronts of total energy consumed vs. total utility earned for the real historical data set (data set 1) for different initial seeded populations through various number of iterations. The circled region represents the solutions that earn the most utility per energy spent. The y-axis values are shared across subplots and while the x-axis values are specific to each subplot.

In the 100,000 iteration subplot, the region highlighted by the circle represents the solutions that earn the most utility per energy consumed. This circle does not represent the best solution in the front, because all solutions along the front are best solutions, each representing a different trade-off between utility and energy. The system administrator may not have energy to reach the circled solution, or may be willing to invest more energy for more utility. To the left of this region, the system can earn relatively larger amounts of utility for relatively small increases in energy. To the right of this region, the system could consume a relatively larger amount of energy but would only see a relatively small increase in utility.

A system administrator can use this bi-objective optimization approach to analyze the utility-energy trade-offs for any system of interest, and then set parameters, such as energy constraints, according to the needs of that system. These energy constraints could then be used in conjunction with a separate online dynamic utility maximization heuristics.

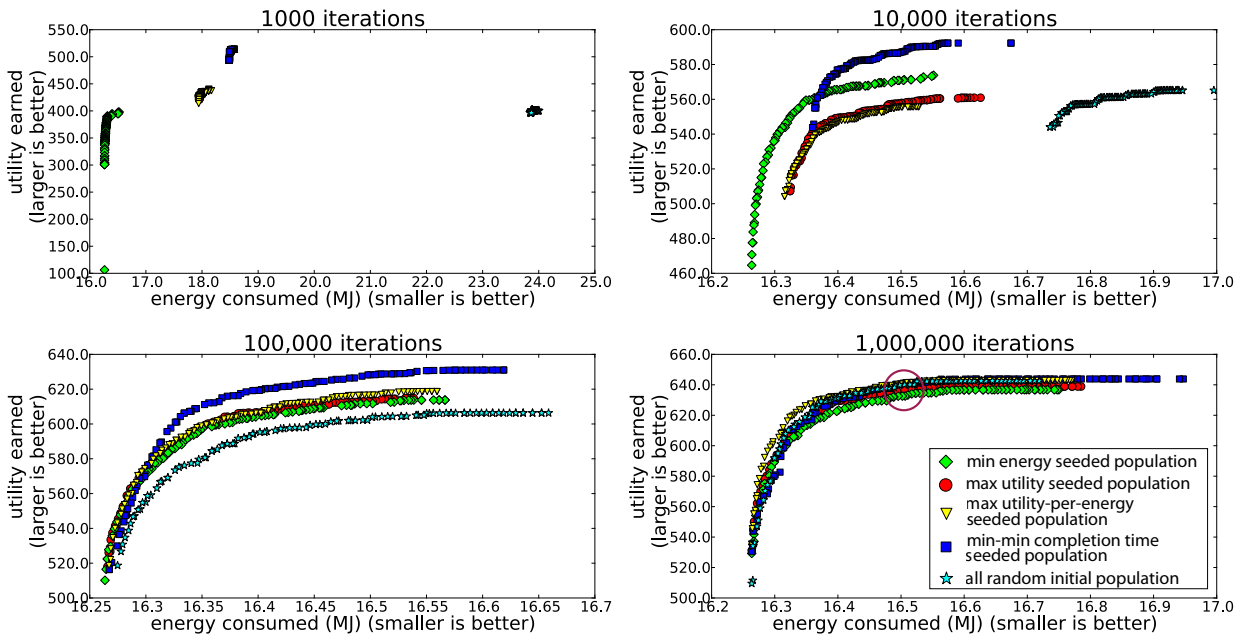


FIGURE 5.4. Pareto fronts of total energy consumed vs. total utility earned for the data set containing 1000 tasks (data set 2) for different initial seeded populations through various number of iterations. The circled region represents the solutions that earn the most utility per energy spent. Both the y-axis and x-axis values are specific to each subplot.

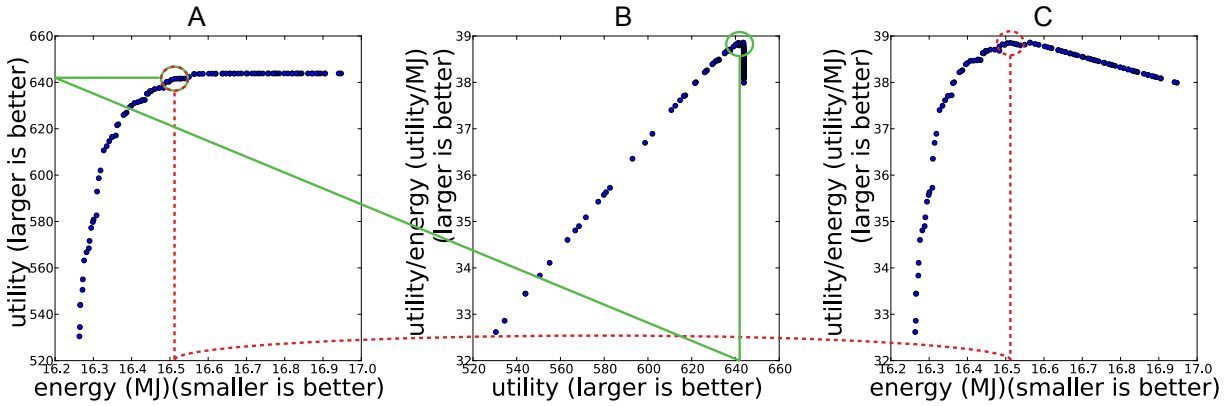


FIGURE 5.5. Subplot A shows the Pareto front through 1,000,000 iterations for the "max utility-per-energy" seeded population. The circled region represents the solutions that earn the most utility per energy spent. Subplot B provides the utility value that gives the highest utility earned per energy spent, shown by the solid line. Subplot C provides the energy value that gives the highest utility earned per energy spent, shown by the dashed line.

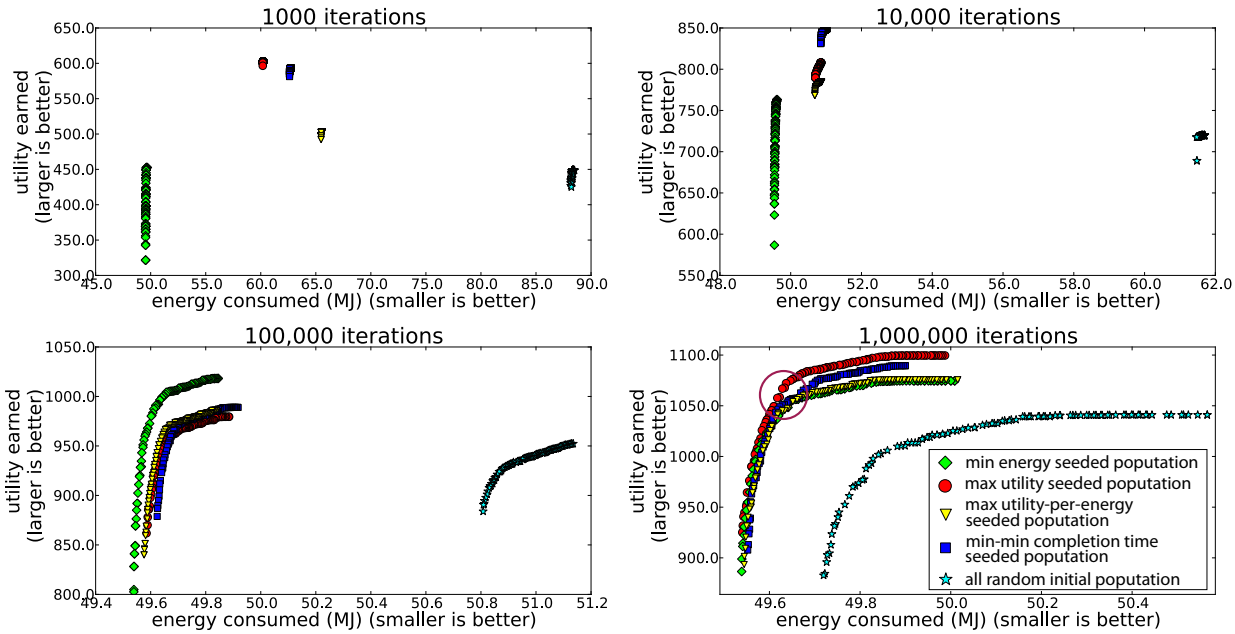


FIGURE 5.6. Pareto fronts of total energy consumed vs. total utility earned for the data set containing 4000 tasks (data set 3) for different initial seeded populations through various number of iterations. The circled region represents the solutions that earn the most utility per energy spent. Both the y-axis and x-axis values are specific to each subplot.

The Pareto fronts in Figure 5.4 illustrate the trade-offs between total energy consumed and total utility earned for the data set that contains 1000 tasks. These fronts were evaluated at 1000, 10,000, 100,000, and 1,000,000 iterations. We increased the number of iterations executed for this data set compared to the real historical data set because we are simulating a larger and more difficult problem.

Examining the top two subplots, we see the effect the initial seeds have on each of the populations. This provides the chance for system administrators to make some interesting observations about their systems. For example, the "min energy" population typically finds solutions that perform better with respect to energy consumption, while the "min-min completion time" population typically finds solutions that perform better with respect to utility earned. This analysis could provide insight into what type of dynamic heuristics could be

used to either maximize utility or minimize energy depending on the needs of a given system. These subplots also show that using smart resource allocation seeds can produce better solutions in a limited number of iterations

In the lower two subplots, we see that all the populations have started to converge towards the same Pareto front. This allows us to find the region containing the solutions that earn the most utility per energy and can provide system administrators with valuable information about the trade-offs between the total energy consumed and the total utility earned of their specific systems. Figure 5.5 illustrates how the maximum utility per energy region is found. Subplot 5.5.A shows the final Pareto front for the "max utility-per-energy" seeded population. Subplot 5.5.B shows a plot of utility earned per energy spent vs. utility while Subplot 5.5.C shows a plot of utility earned per energy spent vs. energy. By locating the "peaks" in both these plots we can find the utility and energy values, respectively, for which utility earned per energy spent is maximized. We can then translate these values onto the Pareto front to find where this region is located. This is shown using the solid lines for utility and the dashed lines for energy.

Finally, Figure 5.6 contains the Pareto fronts for the largest of our three data sets (4000 tasks). This data set is also evaluated at 1000, 10,000, 100,000, and 1,000,000 iterations. Due to the larger size of this problem, it takes more iterations for the Pareto fronts to start converging. This allows us to see the benefit of using the seeds in the initial populations over the all random initial population. In all cases, our seeded populations are finding solutions that dominate those found by the random population. This occurs because the random initial population has to rely on only crossover and mutation to find better solutions, whereas the seeded populations have the advantage of a solution that is already trying to make smart resource allocation decisions.

Similar to the first two data sets, we see that Pareto fronts for this data set also exhibit a region where the amount of utility earned per energy spent is maximized. This is the location where the system is operating as efficiently as possible, and can help guide system administrators in making decisions to try and reach that level of efficiency for their systems.

5.7. Conclusions

Rising costs of energy consumption and the push for greater performance make the need for energy-efficient computing very important as HPC continues to grow. To begin computing in an energy-efficient manner, system administrators must first understand the energy and performance characteristics of their systems. In this work, an analysis framework was provided for investigating the trade-offs between total utility earned and total energy consumed. A method for creating a synthetic data set that preserves the heterogeneity characteristics of a data set constructed from real historical data was developed.

It was shown that by using the NSGA-II one can create well defined Pareto fronts and it was analyzed how using different seeding heuristics within the initial populations affected the evolution of the Pareto fronts. Finally, the method was tested using three data sets. The first data set contained only real data gathered from online benchmarks. The other two data sets contained synthetic data created from the real data to simulate a larger computing system. These two data sets differed in the number of tasks that were required to execute. Our method successfully illustrates the trade-offs between energy consumption and utility earned for all three data sets.

In summary an analysis framework was designed that: 1) provides the ability to create synthetic data sets that preserve the heterogeneity measures found from real data sets, 2) provides the ability to take traces from any given system and then use our resource allocation

heuristic and simulation infrastructure to plot and analyze the trade-offs between total utility earned and total energy consumed, 3) find the region of the Pareto front where a given system is operating as efficiently as possible, and 4) show the effect different genetic algorithm seeds have for various systems, and how using seeds can create populations that dominate completely random populations. Possible directions for future research are presented in Chapter 9.

CHAPTER 6

Bi-objective Optimization Between Makespan and Energy Consumption[1]

6.1. Introduction

As large computing systems (e.g., supercomputers, clusters, datacenters) have increased in size and performance, the costs of operating these systems have increased as well. A significant portion of these costs can be attributed to the amount of energy that is required to run these systems. Between the years 2000 and 2006 the energy consumption more than doubled for high performance computing (HPC) systems, resulting in servers and datacenters accounting for approximately 1.5% of the total United States energy consumption for that year [57]. This amounts to approximately 61 billion kWh, or $4.5 billion in electricity costs. Energy consumption by HPC systems has continued to increase; from 2005 - 2011 the electricity consumption of these systems has increased by 56% worldwide [58].

To battle the rising costs of energy consumption, it is essential for HPC systems to be energy-efficient. This focus on energy-efficiency must have as little impact to performance as possible. Unfortunately, the goals of saving energy and achieving high performance often conflict with each other. To understand and illustrate the trade-offs between energy consumption and computing performance, this dilemma is modeled as a bi-objective optimization problem. When a problem has multiple objectives, it is often the case that there is not just one single optimal solution, but rather a set of optimal solutions. With this research,

[1]This work appeared in [50] and [59]. This work was supported by a National Science Foundation Graduate Research Fellowship, and the National Science Foundation under grant number CNS-0905399, and by the Colorado State University George T. Abell Endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research also used the CSU ISTeC Cray System supported by NSF Grant CNS-0923386.

a method for developing a set of "Pareto"optimal solutions is provided that not only illustrate the trade-offs between energy consumption and performance for a specific computing system, but also allows the system manager to select a solution that fits the system needs and goals.

In this research, it is examined how utilizing different resource allocations (i.e., mapping of tasks to machines) on a given system can allow one to analyze the trade-offs between energy consumption and performance for that system. The current state of the art resource managers, such as MOAB, are unable to reasonably determine the trade-offs between performance and energy based on experience with it in cluster computing environments.

A heterogeneous distributed computing environment used to execute a workload consisting of a bag of tasks is modeled. In such an environment, a task may have different execution and power consumption characteristics when executed on different machines. This behavior requires one to explore different resource allocations to optimally manage the energy consumption and performance of the system. A resource allocation is defined to be a complete mapping of all the tasks in the bag to the machines. The competing nature of minimizing energy consumption and increasing system performance allows this problem to be modeled as a bi-objective optimization problem.

Many bi-objective optimization algorithms exist, such as those found in [73], that can be adapted and used to produce resource allocations for this problem. The solutions produced by such algorithms are used to create graphical representations that allow one to analyze performance and energy trade-offs. Plots are produced that show general trends between performance and energy consumption, as well as more detailed graphs that allow one to analyze how different allocations use the system on a machine-by-machine basis. Analysis of these graphs can help system administrators find allocations that will allow the system

to run at a specified energy/performance level as well as identify inefficiencies in their systems. Additionally, system administrators may desire to simulate the effect and observe the performance and energy consumption implications of adding or removing machines to the system. This could lead to the design of more efficient and cost effective computing systems.

The trade-offs between energy consumption and performance for three different environments are examined. Each environment is based on a set of real machines and real tasks. By analyzing numerous resource allocations, it is shown that for each environment the behavior of the systems can differ greatly, allowing system administrators to select a resource allocation that best fits the needs of their system.

In this chapter the following contributions are made: (a) address the concern of energy efficient computing by modeling the resource allocation problem as a bi-objective optimization problem between minimizing energy consumption and maximizing performance (minimizing makespan), (b) adapt a well-known multi-objective genetic algorithm to the domain of data center task scheduling, (c) show that by using different resource allocations, one can greatly affect the energy consumption and performance of a heterogeneous computing system, (d) construct a set of "Pareto"[72] optimal solutions that can be used to illustrate the trade-offs between system performance and energy consumption, as well as allowing data center managers to select appropriate resource allocations to meet the needs of the specific system, (e) perform a machine-by-machine analysis of how different resource allocations can affect the perfomance and energy consumption of a given system, and (f) provide an analysis approach that can identify both energy efficient and energy-inefficient machines, allowing system administrators to use this knowledge to help build and manage their systems.

The remainder of the chapter is organized as follows. Related work is discussed in Section 6.2. In Section 6.3, the bi-objective optimization problem is described. The system model

is explained in Section 6.4. The experimental setup and results for two separate studies are presented in sections 6.5 and 6.6. Finally, the conclusions given in Section 6.7.

6.2. Related Work

Several prior efforts have examined bi-objective resource allocation problems in large computing environments.

The bi-objective genetic algorithm NSGAII [61] is adapted for use within the resource allocation domain in [59] and [45]. System-level analyses are performed, specifically looking at the trade-offs of energy and makespan [59] or energy and utility [45]. Our current work performs an in-depth analysis on a machine-by-machine level with a realistic system model.

A bi-objective heterogeneous task scheduling problem between makespan and reliability is presented in [62] and [63]. Instead of reliability, our work investigates the trade-offs between makespan and energy consumption with a machine-by-machine allocation analysis.

In [64], the authors solve a bi-objective optimization between makespan and robustness for a heterogeneous scheduling problem. Solutions are created using a weighted sum simulated annealing heuristic, where one run of the heuristic produces a single solution. In our work we are concerned with multiple solutions, and analyzing how their allocations change based on their location in the search space.

A bi-objective flowshop scheduling problem between makespan and total tardiness is modeled in [65]. Solutions are created using a Pareto-ant colony optimization approach. While we could use methods such as this Pareto-ant approach, the focus of this chapter is on the resulting allocations, not how they are created.

The authors of [74] model a homogeneous job-shop scheduling problem between makespan and energy consumption. We are interested in analyzing the behavior of systems that consist of heterogeneous machines, which significantly changes the problem and solution space.

An energy-constrained heterogeneous task scheduling problem is examined in [8]. In this environment, the energy constraint is realized by modeling devices with limited battery capacity in an ad-hoc wireless network. In our work, we are not directly concerned with an energy constraint, though by analyzing different solutions from a Pareto front, a solution could be picked that meets an energy constraint if it was needed.

In [67], the authors try to minimize energy consumption while trying to meet a makespan robustness constraint. Because there is a constraint on the makespan robustness, this is not a bi-objective optimization problem, and does not involve the type of machine-by-machine analysis that we preform.

There are many environments that can be modeled as dynamic resource allocation problems. One such environment is [10], where the system must complete as many tasks as possible by their individual deadlines while staying within the energy budget of the system. This environment does not perform a machine-by-machine analysis to to investigate the trade-offs between energy and makespan.

6.3. Bi-Objective Optimization Using Genetic Algorithms

6.3.1. Overview. It is common for many real-world problems to contain multiple goals or objectives. Often, these objectives work against each other, as optimizing for one objective can negatively impact another objective. This is the case in our research, because it is important for HPC systems to be concerned with both lowering energy consumption as well as increasing overall system performance. In general, resource allocations using more energy

will allow one to achieve greater performance, while resource allocations trying to conserve energy will cause the system to have slower performance. In Section 6.3.2, we describe how to determine which solutions should be considered when trying to solve a bi-objective optimization problem. We then briefly discuss the genetic algorithm we have adapted to solve our specific problem in Section 6.3.3.

6.3.2. Determining Solutions to a Bi-Objective Optimization Problem. When multiple objectives are present within a problem, it is often the case that there does not exist a single global optimal solution, but rather a *set* of optimal solutions. There is no guarantee one can find the exact solutions within this optimal set, so instead we try to find a set of solutions that are as close to the optimal set as possible. We will call this set of solutions the set of Pareto optimal solutions [72]. This set of Pareto optimal solutions can be used to construct a Pareto front that illustrates the trade-offs between the two objectives.

To understand what it means for a solution to be part of the Pareto optimal set, we illustrate the notion of solution dominance. Dominance is defined as one solution being better than another solution in at least one objective, and better than or equal to in the other objective. To help explain what it means for one solution to dominate another, please refer to Figure 6.1. Figure 6.1 shows three potential solutions. The objectives are to minimize energy (along the x-axis), and to minimize makespan (along the y-axis). Let us first examine the relationship between solutions A and B. From the figure we can see that solution B is dominated by A because A uses less energy and has a smaller makespan. Likewise, any solution residing within the upper right (green) shaded region would also be dominated by A. Next, consider solutions A and C. We cannot claim either solution dominates the other because A uses less energy than C, but C has a smaller makespan than A. Thus, for this example, both A and C are solutions in the Pareto optimal set and form the Pareto front.

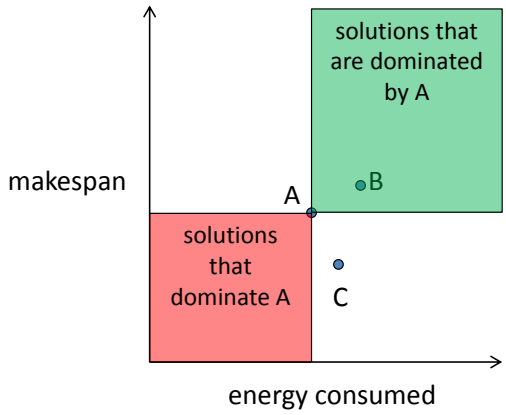


FIGURE 6.1. Illustration of solution dominance for three solutions: A, B, and C. Solution A dominates solution B because A has lower energy consumption as well as a lower makespan. Neither solution A nor C dominate each other because A uses less energy, while C has a lower makespan.

Finally, solution A would be dominated by any solution residing within the lower left (red) shaded area.

6.3.3. NONDOMINATED SORTING GENETIC ALGORITHM II. The Nondominated Sorting Genetic Algorithm II (NSGA-II) was designed for solving bi-objective optimization problems [61] and has been adapted to create Pareto fronts of makespan and energy. A full description of the NSGA-II algorithm can be found in Section 5.4.4.

6.4. SYSTEM MODEL

6.4.1. MACHINES. We model a heterogeneous suite of M machines, where each machine is one of MT machine types. Because we are modeling a heterogeneous system, machine type A may be faster for some tasks than machine type B, but may be slower for other tasks [35]. Machines are also heterogeneous in power consumption. We assume that each machine can only execute a single task at a time, similar to the Colorado State University ISTeC Cray [46]. Once a machine finishes executing all of its assigned tasks it shuts down, and no longer consumes any energy.

6.4.2. Workload. We model a workload environment where we have a bag of $\underline{T}$ tasks, and each task belongs to a given task type. Every task is known before the schedule is created. Due to the heterogenous nature of the system, each task type i executing on machine type j will have known execution (Estimated Time to Compute (ETC)) and power consumption (Estimated Power Consumption (EPC)) characteristics denoted as ETC(i,j) and EPC(i,j). Tasks of the same task type have the same ETC and EPC characteristics. In resource allocation, it is common to assume the availability of such characteristics (e.g. [75, 12, 15, 13, 14]). These values may be taken from historical sources ([13, 15]) or may be constructed synthetically for simulation purposes ([76, 35]).

Finally, to obtain the estimated energy consumed (EEC) of a task of type τ on a machine μ we take the product of the execution time and the estimated power consumption, as shown below.

$$EEC[\tau, \mu] = ETC[\tau, \mu] \times EPC[\tau, \mu] \tag{15}$$

6.4.3. Objective Functions.

6.4.3.1. *Makespan.* One objective we are trying to optimize is makespan, which is the total amount of time it takes for all the tasks in the batch to finish executing across all machines. When optimizing for makespan the goal is to minimize the makespan. For a given resource allocation, calculating the makespan of the system requires that we first determine the finishing time of each machine.

To calculate the finishing time of a machine we let the set T_m represent all the tasks in T that were allocated to machine m, where $t_m \in T_m$. Let the function $\Upsilon(t_m)$ return the task type that task t_m belongs to, and let the function $\Omega(m)$ return the machine type to which machine m belongs. We then calculate the expected finishing time of machine m denoted as

F_m, with the following equation

$$F_m = \sum_{\forall t_m \in T_m} ETC(\Upsilon(t_m), \Omega(m)). \tag{16}$$

The makespan for a given resource allocation, denoted ρ, can be found from the machine with the maximum finishing time, and is given as

$$\rho = \max_{\forall m \in M} F_m. \tag{17}$$

6.4.3.2. *Energy Consumption.* The other objective we will optimize for is energy consumption. For a given resource allocation, the total energy consumed is the sum of the energy consumed by each task to finish executing. Recall that the amount of energy consumed by a task is dependent upon the machine on which that task is executing. Therefore, the total energy consumed for a resource allocation, denoted E, can be found as

$$E = \sum_{\forall t_m \in T_m, \forall m \in M} EEC[\Upsilon(t_m), \Omega(m)]. \tag{18}$$

6.5. Investigation One

6.5.1. Experimental Setup.

6.5.1.1. *Simulation Environment Parameters.* To construct a Pareto front and illustrate the trade-offs between makespan and energy consumption, we conducted numerous simulation trials. For each trial, the number of tasks to execute was set to 1000, with 50 different task types. The number of machines used throughout the simulations was set to 50, with 10 different machine types. The number of tasks per task type and number of compute nodes per compute node type were randomly assigned, and could change from trial to trial.

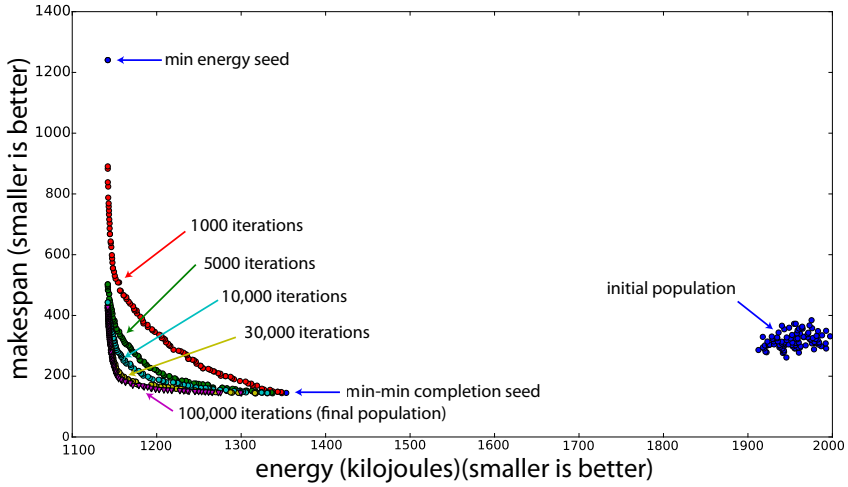


FIGURE 6.2. Pareto fronts showing the trade-offs between energy consumption and makespan. Shows the evolution of the solutions through number of iterations completed

The ETC values were obtained using the Coefficient of Variation (COV) method from [35], which allows us to model a heterogeneous set of machine types and task types. For our simulations, the mean execution time for the tasks was 10 seconds, and the variance amongst the tasks was 0.1, while the variance amongst the machines was 0.25. These parameters allowed us to model a heterogeneous set of compute nodes.

The EPC values were constructed in a similar manner as the ETC values. Specifically, the mean power consumption for the tasks was 200 watts, and the variation amongst tasks was 0.1, while the variance amongst the machines was 0.2.

For each trial, the genetic algorithm consisted of 100 chromosomes. In the initial population, we used 98 randomly generated chromosomes, and two chromosomes generated using two heuristics based on approaches taken from literature, as discussed below.

6.5.1.2. *Seeding Heuristics.* The goal of the seeding heuristics are to provide the genetic algorithm with initial solutions that try to optimize the objectives. These seeds can help

guide the genetic algorithm towards better solutions faster than an all-random initial population. We chose to implement two greedy heuristics, min energy and min-min completion time, based on concepts found in [19, 54, 9]. The execution times of the greedy heuristics are negligible compared to the NSGA II. Utilizing these seeds in the initial population does not negatively affect the computation time of the NSGA II.

Min Energy. Min energy is a single stage greedy heuristic that maps tasks to machines to minimize energy consumption. The heuristic selects a task from the batch and places that task on to the machine that has the smallest energy consumption. For this heuristic, the order in which tasks are mapped to machines does not matter. This heuristic creates a solution that will have the minimum possible energy consumption. For a solution to be more efficient, it must have a smaller makespan.

Min-Min Completion Time. Min-min completion time is a two-stage greedy heuristic that maps tasks to machines to minimize the makespan of the system. During each iteration of the heuristic, one task gets mapped to the machine that provides the minimum completion time. One iteration consists of two stages. In the first stage, every unmapped task finds the machine that minimizes completion time. In the second stage, the heuristic selects the task and machine pair from the first stage that has the smallest overall completion time and assigns that task to that machine. This continues until there are no more tasks to map. There is no guarantee that the solution created by this heuristic represents the absolute lowest makespan of the system, so better solutions can potentially improve in both makespan and energy consumption.

6.5.2. Results. Throughout this section, we will only be discussing the results from one simulation trial. We have confirmed that the findings and trends for this trial hold for the other trials we ran. In Figure 6.2, we show the evolution of the solutions through the

number of NSGA-II iterations completed. It is important to note for genetic algorithms, as we increase the number of iterations, the genetic algorithm will in general find new and better solutions; some solutions may remain a member of the Pareto front as we increase the iterations. Each point in Figure 6.2 represents a complete resource allocation. The set of points corresponding to a given number of iterations form the Pareto front. These points are obtained from the genetic algorithm running through that number of iterations. We see that as the genetic algorithm runs for more iterations, the Pareto fronts are converging towards the lower-left corner. This makes sense because we are minimizing makespan as well as energy consumption. We can also see that for this size problem there is very little improvement to the Pareto front after 30,000 iterations. The size of the problem as well as using the two seeds help the solutions converge in a relatively short number of iterations. Also, observe that both of the seeds provide good starting solutions for the genetic algorithm to evolve from relative to the rest of the initial population.

Although it is useful to see how the solutions evolve over time, the most important information to take away from Figure 6.2 are the trade-offs between makespan and energy consumed. Figure 6.3 shows a blown-up plot of the final Pareto front from Figure 6.2. There are a number of points we can learn from the final Pareto front shown in Figure 6.3. One such point is circled in red. We can see that around this point there is a definite and visible "knee" in the front. To the left of the knee, small increases in energy consumption result in large decreases in makespan. To the right of the knee we see the opposite, small decreases in makespan result in large increases in energy. Given the information provided in this Pareto front, it is then up to the system manager to select which region of the curve to operate in based on the individual system needs.

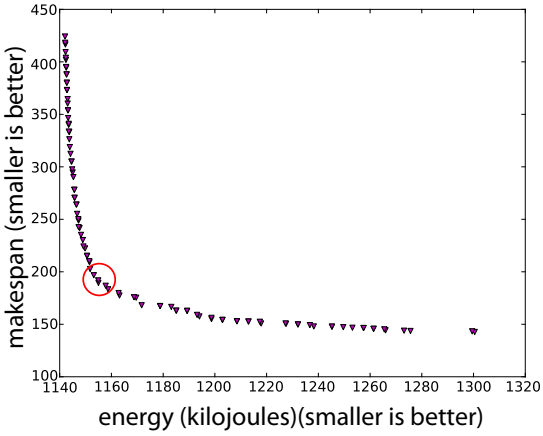


FIGURE 6.3. The final Pareto front after 100,000 iterations showing the trade-offs between energy consumed and makespan.

To further understand how solutions in the Pareto front differ from one another, we analyzed the individual finishing times and energy consumptions for the 50 machines at five points along the final Pareto front. The five points were the two endpoints of the front, the middle point, and the two points between the middle point and each endpoint, as shown in Figure 6.4 and Figure 6.5. The results for machine finishing times are shown in Figure 6.4, while the results for machine energy consumption are shown in Figure 6.5.

First, we consider Figure 6.4, which focuses on the finishing times of each machine. Each of the five subplots (b-f) in the figure represents a solution from the Pareto front. On the x-axis of the plots we have the machines sorted by finishing time in descending order, this ordering is different from subplot to subplot. On the y-axis we have the actual finishing time of each machine. In Figure 6.4.f, we have the solution that provides the lowest makespan. As we can see, the finishing times for all the machines are evenly balanced; this allows the makespan to be small since no one machine is doing a lot more work than the others. As we move left along the Pareto front selected points in Figure 6.4.a (minimizing energy) we see that the solutions become more and more unbalanced with respect to machine finishing times going from Figure 6.4.f to Figure 6.4.e to Figure 6.4.d, etc. This is because each

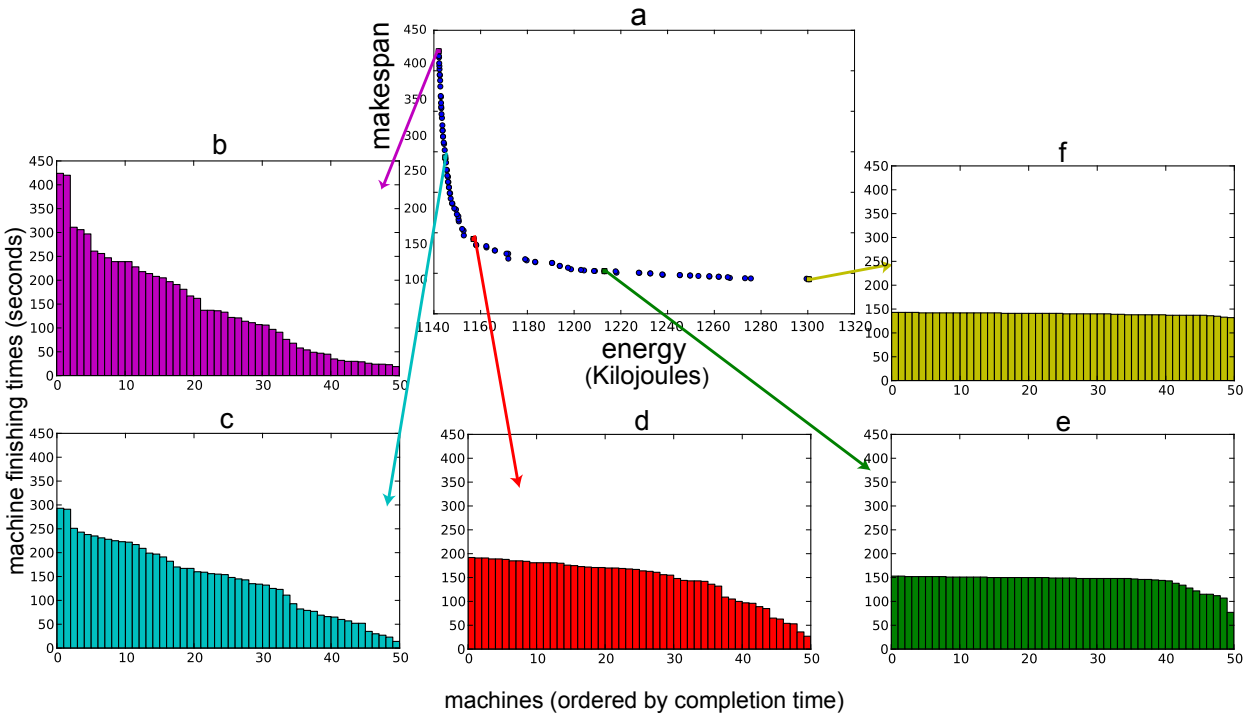


FIGURE 6.4. Finishing time of the 50 machines in descending order of finishing time for five solutions from the Pareto front. The y-axis contains the same range of machine finishing times from plot to plot. Subfigure "a" is the same as Figure 6.3 and has different axis labels from the other subplots. Each subplot b-f has a different ordering of the machines along the x-axis.

task type has an affinity for a specific machine type that minimizes that task type's energy consumption.

If we now consider the plots of machine energy consumption in Figure 6.5 which focus on energy consumption, we see similar trends as before. This time the machines are ordered in descending order based on energy consumption. Figures 6.5.f and 6.5.e are more balanced in terms of energy consumption amongst the machines. It is interesting to note that both these figures are more unbalanced than their counterparts from Figure 6.4, this because the machines have varying degrees of energy efficiency. This characteristic is also the reason that Figure 6.5.a is so unbalanced, most of the tasks are being executed on the most energy efficient machines, because this area of the Pareto front focuses on trying to minimize energy.

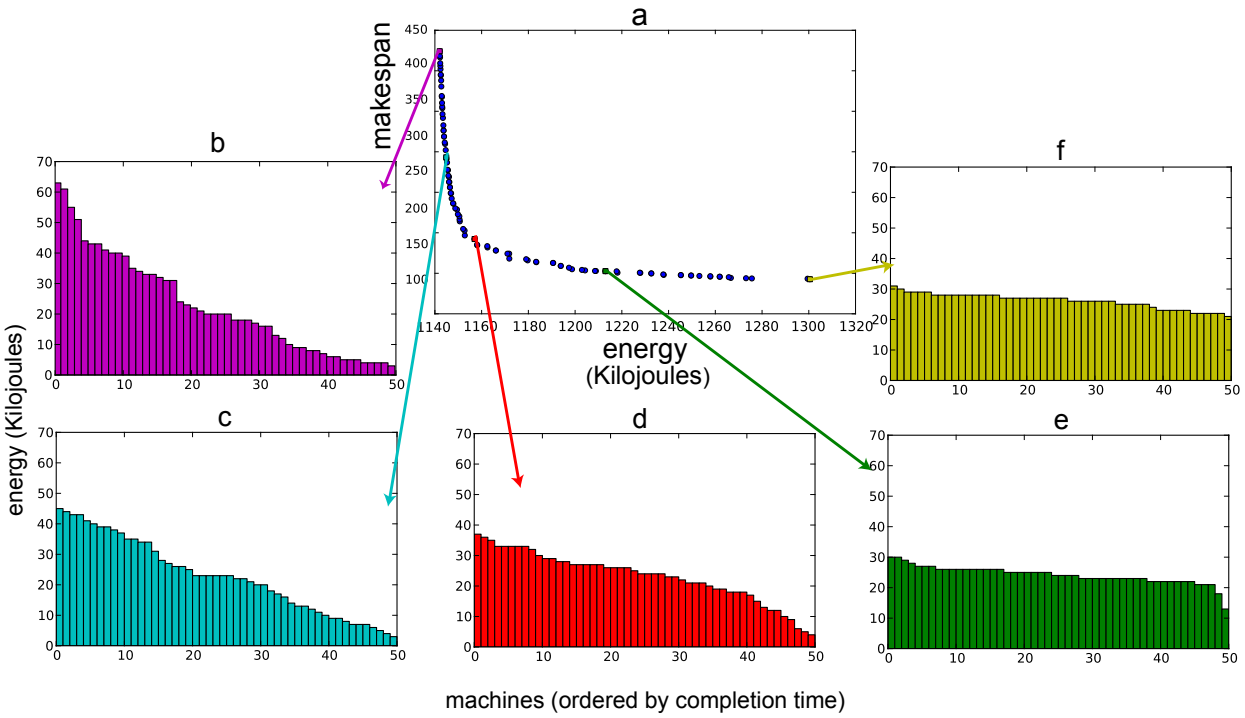


FIGURE 6.5. Energy consumption of the 50 machines in descending order of energy consumed for five solutions from the Pareto front. The y-axis contains the same range of machine energy consumption from plot to plot. Subfigure "a" is the same as Figure 6.3 and has different axis labels from the other subplots.Each subplot b-f has a different ordering of the machines along the x-axis.

With the information provided by the Pareto fronts as well as the plots showing the completion time and energy consumption of individual machines, a system manager will be able to analyze the trade-offs between energy consumption and makespan. The system manager can then make a scheduling decision based on the needs of the computing system.

6.6. Investigation Two

6.6.1. Experimental Setup.

6.6.1.1. *Datasets.* To accurately model the relationships between machine performance and energy consumption in the ETC and EPC matrices, we used the method outlined in Chapter 5 ([45]) where a dataset consisting of five applications executed on nine machines [69]

TABLE 6.1. Machines Types (designated by CPU)

1	AMD A8-3870k
2	AMD FX-8159
3	Intel Core i3 2120
4	Intel Core i5 2400S
5	Intel Core i5 2500K
6	Intel Core i7 3960X
7	Intel Core i7 3960X @ 4.2 GHz
8	Intel Core i7 3770K
9	Intel Core i7 3770K @ 4.3 GHz

TABLE 6.2. Task Types

1	C-Ray
2	7-Zip Compression
3	Warsow
4	Unigine Heaven
5	Timed Linux Kernel Compilation

is used to create a larger synthetic dataset. The synthetic data set resembles the original data set in terms of heterogeneity characteristics, such as the coefficient of variation, skewness, and kurtosis [70]. The original machine types (designated by CPU) are listed in Table 6.1 and the original task types are listed in Table 6.2. The original data contained both execution times and total system power consumption (measured at the outlet) for each task on each machine. Each machine used 16GB of memory, a 240GB SSD, and ran Ubuntu 12.04 (but had different processors).

6.6.1.2. *Experiments.* We considered three test environments each with 36 machines. The first test environment only used machine types 1 and 2 and there were 18 machines of each type in the system. The second test environment utilized machine types 1-6, and there existed six machines per type. Finally the third test environment consisted of the full set of nine machine types with four machines belonging to each type. The bag of tasks for each test environment was identical and consisted of 1000 tasks distributed among 30 task types (the five original task types and 25 synthetic task types).

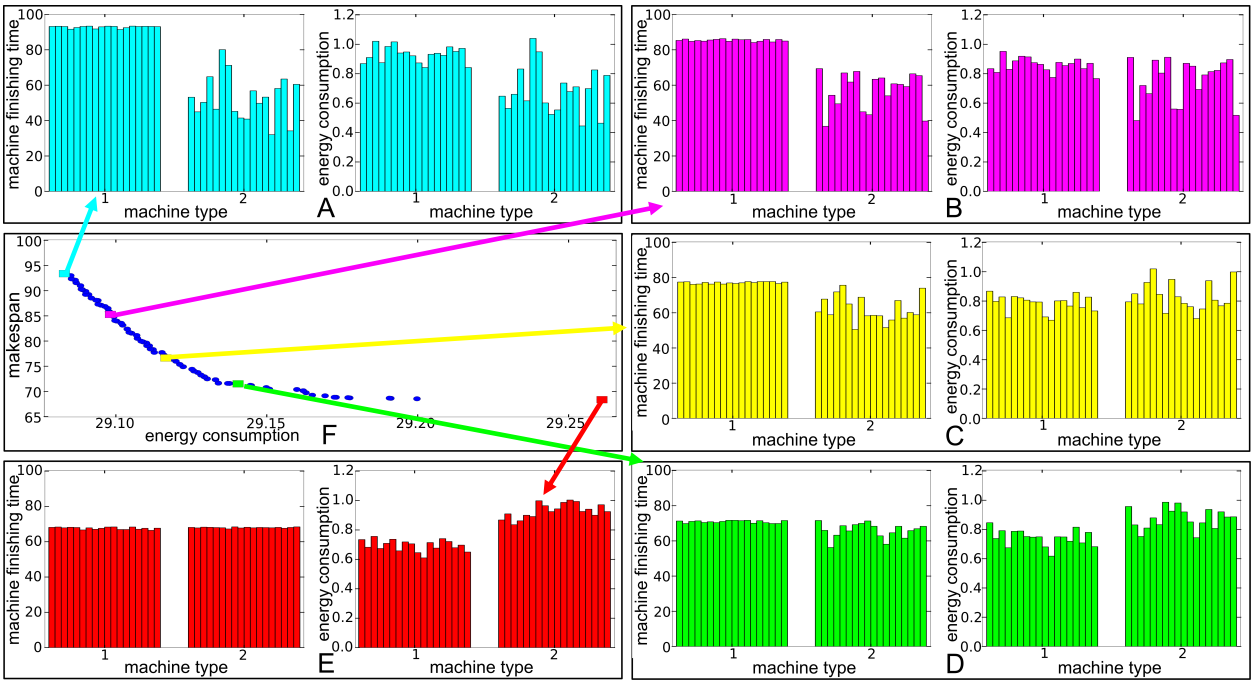


FIGURE 6.6. Pareto front (F) and five resource allocations (A-E) for an environment with two machine types. The Pareto front illustrates the trade-offs between energy consumption in megajoules (x-axis) and makespan in minutes (y-axis). In A-E, the left graph shows the completion of each machine in minutes (y-axis) in the system. The right graph shows the energy consumption of each machine in megajoules (y-axis). In both graphs the machines are grouped by machine type (x-axis).

6.6.2. RESULTS. In Figs. 6.6- 6.8 we present the results of our experiments for the two machine type, six machine type, and nine machine type environments, respectively. In each fig., the subfig. "F" shows the Pareto front for each environment, where the x-axis is the total system energy consumption measured in megajoules(smaller is better) and the y-axis is the makespan of the system measured in minutes (smaller is better). Each individual marker in these plots represents a complete resource allocation. In each of these Pareto fronts, we see that makespan decreases (e.g. system performance increases) as the energy consumption of the system increases. This is consistent with the results from [59] and [45]. To better understand why this trend occurs, we analyzed five separate resource allocations from the Pareto front (the square markers in each of the Figs. 6.6F, 6.7F, and 6.8F).

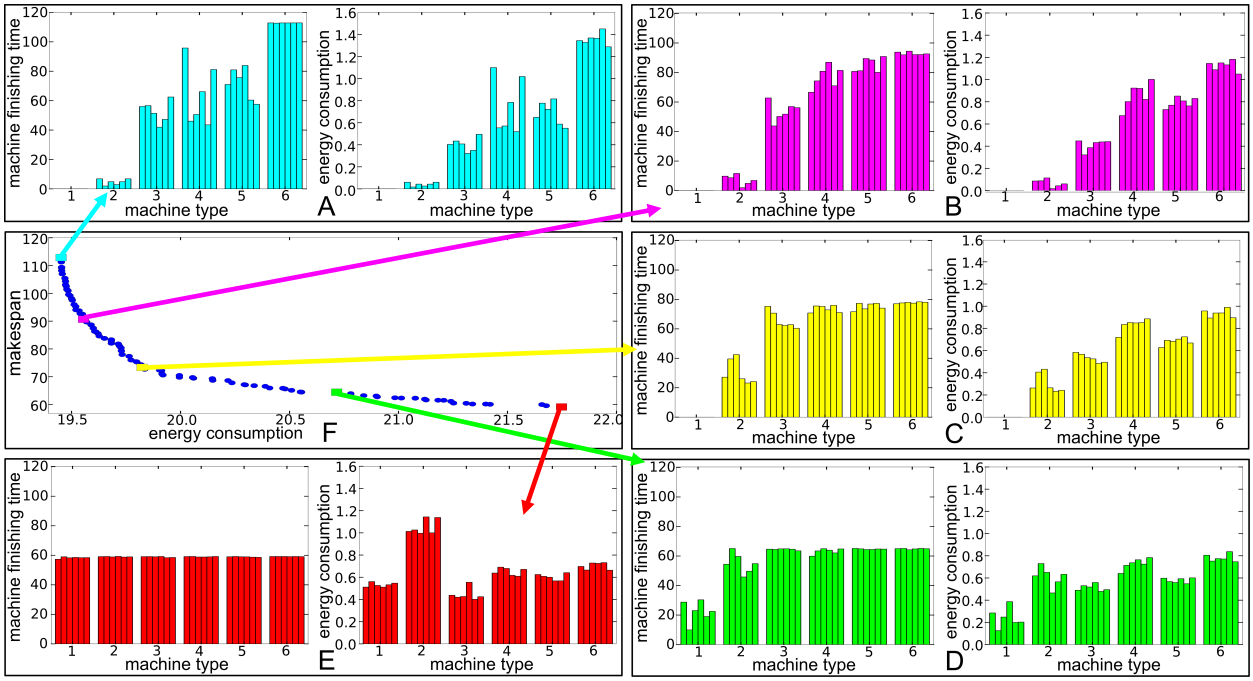


FIGURE 6.7. Pareto front (F) and five resource allocations (A-E) for an environment with six machine types. The Pareto front illustrates the trade-offs between energy consumption in megajoules (x-axis) and makespan in minutes (y-axis). In A-E, the left graph shows the completion of each machine in minutes (y-axis) in the system. The right graph shows the energy consumption of each machine in megajoules (y-axis). In both graphs the machines are grouped by machine type (x-axis).

For each of our selected resource allocations for each environment, we plotted the completion time and energy consumption of each machine, and grouped the machines by machine type as seen in subfigs. A-E for each environment. Subfigs. A-E range from illustrating the minimum energy consumption allocation in subfig. A to the minimum makespan allocation in subfig. E. For the minimum energy allocations (subfig. A), each task ends up being assigned to a machine that is part of the machine type that executes that task with the least amount of energy.

In the A subfigs. (in all three environments), we find that for each environment there is one machine type that has a longer finishing time than the other machine types (the left graph in the subfig.), and thus determines the makespan for this solution. This occurs

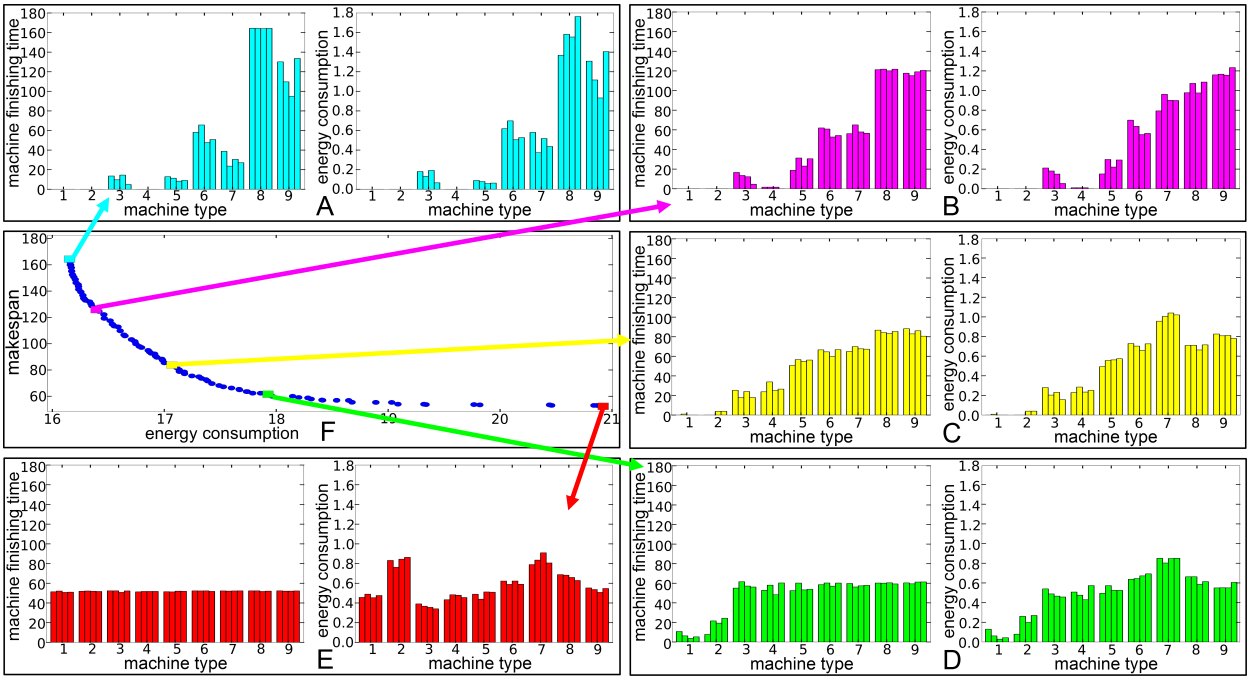


FIGURE 6.8. Pareto front (F) and five resource allocations (A-E) for an environment with nine machine types. The Pareto front illustrates the trade-offs between energy consumption in megajoules (x-axis) and makespan in minutes (y-axis). In A-E, the left graph shows the completion of each machine in minutes (y-axis) in the system. The right graph shows the energy consumption of each machine in megajoules (y-axis). In both graphs the machines are grouped by machine type (x-axis).

because that machine type has more tasks for which it is the minimum energy machine type (implying it is a more energy-efficient machine type), thus those tasks will prefer to run on machines of that type, forcing the completion time for those machines to increase. There can exist many allocations that minimize energy consumption, therefore to be in the Pareto front, a minimum energy allocation must dominate by lowering the makespan of the system. The resource allocation heuristic accomplishes this by decreasing the finishing times of machines in the longest finishing time machine type. This results in balanced finishing times for machines of that type as can be seen in the subfigs. The other machine types have unbalanced finishing times among their machines because they do not have any effect on the makespan of the system. Another interesting observation is that in the environments

with six and nine machine types (Subfigs. 6.7A and 6.8A) there exist machines that do not execute even a single task. This is because these machine types are not the minimum energy machines for any task.

Examining the subfigs. from A to E, we find the completion times of all the machines start to become balanced. This is because to lower the system makespan, a resource allocation must distribute the tasks to all the machines so that one machine (or machines in a machine type) is not forced to execute significantly more tasks than the other machines. Hence, working through the Pareto front (from left to right) the machine completion times become more balanced (lower makespan) by forcing tasks to run on machine types that consume higher amounts of energy. This becomes most apparent in the E subfigs., where the machines have balanced finishing times (left graph) but their energy consumptions (right graph) are not balanced. This shows that as makespan decreases, the amount of energy consumed increases.

It is important to note in Figs. 6.7 and 6.8 that even as the allocations are decreasing makespan, there are still certain machines that execute very few tasks, and it is not until the lowest makespan allocation that they execute a comparable number of tasks as the other machines. This occurs because the trade-off in decreasing makespan versus the amount of energy the system would consume is not large enough to warrant using these energy-inefficient machines.

By examining Pareto fronts and various resource allocations from within those Pareto fronts, system administrators can gather important information about the operation of their systems. This includes finding machines and machine types that are energy-inefficient. With this knowledge, a system administrator may decide to leave these machines off to save energy unless it is absolutely necessary to finish a workload as fast as possible. Additionally, they

can see which machine types are being utilized the most and make future purchasing decisions based on this information.

Finally, this study provides and example of how system administrators can perform "what-if" analyses. For example, what if we add more machine types to the system, what if we add more machines of a specific machine type, or what if we turn off certain machines? All of these scenarios could be simulated and then analyzed by the system administrator to help them decide how to best manage their system. We illustrate the power of these type of questions by comparing our three test environments against one another. We see that as we increase the number of machine types in the environment, we are able to both lower the makespan and have a smaller total energy consumption for the system. There are many reasons this may occur, the most straightforward is that more powerful and energy efficient machine types are added. Another reason is that additional machine types may increase the heterogeneity of the system, resulting in task-machine affinity being exploited. We are also able to see that it may be better to invest in machines that are of types 8 and 9 as they are the machines that execute the most tasks in the most energy efficient manner, while it may be best to not use machine types 1 or 2 at all as they both consume more energy than the other machine types.

The analyses performed in this work cannot be done by evaluation of only the ETC and EPC characteristics, rather, they require a more comprehensive analysis of the complex interaction between the workload, machines, two objectives.

6.7. Conclusions

As high performance computing systems continue to become more powerful, the energy required to power these systems also increases. In this chapter a bi-objective optimization

model that can be used to illustrate the trade-offs between the makespan and energy consumption of a system has been developed. Having adapted the nondominated sorting genetic algorithm for use within this domain, successful simulations that provided well defined Pareto fronts were run for two separate studies.

In the first study, a synthetic environment and workload were created, and five different solutions from a final Pareto front were examined. The differences in their makespan and energy consumption were discussed and it showed that to decrease makespan the system had to increase energy consumption. Given this information a system administrator would be able to pick a specific resource allocation from the Pareto front that meets the energy and performance needs of the system.

In the second study, three different environments based on actual machine and application benchmark data were studied. The three environments different in the number of machine types. By analyzing individual resource allocations it is possible to examine how a given system is distributing and executing tasks depending on the performance and energy consumption desired. This investigation can help identify the degree of energy-inefficiency of the machines, as well as allow system administrators to perform "what-if" analyses to determine the effect of adding or removing machines from their systems. Possible future work for these studies can be found in Chapter 9.

CHAPTER 7

Stochastic Multi-Objective Optimization of System Performance and Energy Consumption[1]

7.1. Introduction

Many real world problems contain multiple objectives, and often it is necessary to consider the trade-offs that occur between these objectives when optimizing for one objective over the other. In Chapters 5 and 6, bi-objective optimization between maximizing system utility earned and minimizing system energy consumed in a static, offline, and deterministic heterogeneous environment was performed. Total utilty earned is the sum of utility earned by each task. Total energy consumed is the sum of dynamic energy consumed by each task. In a heterogeneous environment, tasks may have different execution and power consumption characteristics when executed on different machines. A resource allocation is defined to be a complete mapping of tasks to machines, where it is assumed the number of tasks is much greater than the number of machines. One can change the performance and energy consumption of the system by using different resource allocations. To analyze the effect of different resource allocations, a post-mortem analysis is performed by considering the environment in an offline manner. To create the offline environment, a trace is simulated over a specified time period of the modeled environment. This allows one to gather information about the arrival times and number of tasks during this time period. The availability of this information makes this a static resource allocation problem. It was shown in [45] that maximizing

[1]This work was supported by Oak Ridge National Laboratory and their Extreme Scale Systems Center under subcontract number 4000108022, and by a National Science Foundation Graduate Research Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research also used resources of the Extreme Scale Systems Center at Oak Ridge National Laboratory as well as the CSU ISTeC Cray System supported by NSF Grant CNS-0923386.

utility and minimizing energy are conflicting objectives, meaning that to increase the utility earned by the system, the energy consumed typically needs to increase, while decreasing the energy consumed usually results in less utility being earned.

In this chapter, previous studies in Chapters 5 and 6 are expanded by using stochastic distributions for the estimated time to compute (ETC) and average power consumption (APC) matrices. A task's execution time and energy consumption can change depending on the data set being processed, thus, stochastic distributions are used to model these uncertainties in execution time and power consumption. To account for stochastic distributions, two new objective methods were designed and implemented to maximize utility and minimize energy. The first method is based on the expected values of execution time and energy. The second method is based on probabilistic evaluations. In this stochastic environment, a single resource allocation (complete mapping of tasks to machines) can be evaluated multiple times to produce different utility-energy value pairs by sampling the stochastic distributions to generate the task execution times. In this environment, tasks with similar execution requirements and characteristics are part of the same task type, while machines with similar computational capabilities are part of the same machine type. Additionally, machine performance states (P-states), which represent the dynamic voltage and frequency scaling presets of processors that allow the processor to reduce the dynamic power consumption by reducing execution speed are modeled.

The remainder of this chapter is set up as follows. Section 7.2 formally describes the bi-objective optimization problem. The experimental setup is given in Section 7.3. Results are presented in Section 7.4. The conclusions are presented in Section 7.5.

7.2. Bi-Objective Optimization

7.2.1. Overview. In many real world problems, it is important to consider more than one goal or objective. It is often the case that these multiple objectives can directly or indirectly compete with each other. Optimizing for one objective may cause the other objective to be negatively impacted. In this work, we try to maximize utility earned while minimizing energy consumed. We will show in our results that these two objectives do compete with each other. In general, a well-structured resource allocation that uses more energy will earn more utility and one that uses less energy will earn less utility. This relationship occurs because we model utility as a monotonically-decreasing performance measure, that is, the longer a task takes to execute, the less utility it may earn.

In Section 7.2.2, we define the objective functions we are optimizing. Section 7.2.3 describes which solutions should be considered when solving a bi-objective optimization. The genetic algorithm used for this study is briefly explained in Section 7.2.4.

7.2.2. Objective Functions.

7.2.2.1. *Expected Value-Based.*

Maximizing Expected Utility: One objective is maximizing total expected system utility earned. Total expected utility earned is a metric used to measure the performance of a computing system. Utility can be thought of as the amount of useful work a system accomplishes [36]. The amount of utility a task earns is determined by the expected completion time of the task (derived from the task's stochastic completion time distribution), as well as by three parameters: priority, urgency, and utility class. These parameters form what is called a utility function. This function is a monotonically-decreasing function [36].

Priority represents the maximum amount of utility a task could possibly earn, i.e., how important a task is. Urgency represents the rate of decay of utility. Utility class allows the utility function to be separated into discrete intervals. Each interval can have a beginning and ending percentage of maximum utility, as well as an urgency modifier to control the rate of decay of utility.

Figure 7.1 illustrates a sample task utility function. We see that the function is monotonically decreasing. By evaluating the function at different completion times, we can determine the amount of utility earned at those different times. For example, if this sample task finished at time 20, it would earn twelve units of utility, whereas if the task finished at time 47, it would only earn seven units of utility.

Every task in the system is assigned its own utility function. This function returns the utility earned by that task when it completes execution. Tasks that have hard deadlines can be modeled as having their utility decay to zero by the deadline. To optimize for total expected utility, the goal is to maximize the sum of expected utilities earned over all tasks in the system.

Minimizing Expected Energy Consumed: The other expected value-based objective is minimizing total expected energy consumed. For a given resource allocation, the total expected energy consumed is the sum of the expected energy consumed by each assigned task. We first calculate the Expected Energy Consumption (EEC) for a given task on a given machine. This is found by multiplying the expected execution time ($\mathbb{E}[ETC]$) and expected average power consumption ($\mathbb{E}[APC]$) values from the ETC and APC distributions for that task on that machine and P-state. Assume $\Phi(t)$ represents the task type of a given task t, $\Omega(m)$ represents the machine type of a given machine, and p is the P-state to execute

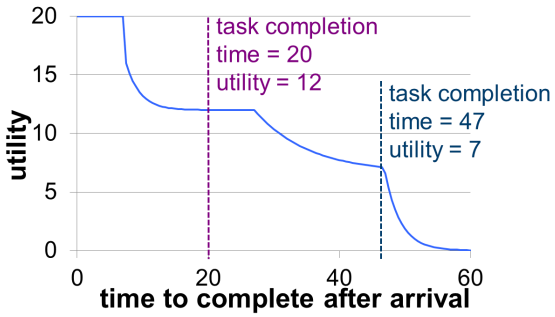


FIGURE 7.1. A task utility function showing utility earned at different completion times.

in. The expected energy consumption of task t on machine m can then be given as

$$EEC[\Phi(t), \Omega(m), p] = \mathbb{E}[ETC[\Phi(t), \Omega(m), p]] \times \mathbb{E}[APC[\Phi(t), \Omega(m), p]]. \tag{19}$$

Let $\underline{T_m}$ be the tasks that execute on machine m. The total expected energy consumed by the system, denoted $\underline{\mathrm{E}}$, is found by

$$E = \sum_{\forall m \in M} \sum_{\forall t_m \in T_m} EEC[\Phi(t_m), \Omega(m), p]. \tag{20}$$

7.2.2.2. *Probabilistic-Based.*

Maximizing Utility at X^{th} Percentile: The first probabilistic-based objective tries to maximize utility. Due to the fact we are using stochastic distributions to model task execution times, a single resource allocation will not always evaluate to the same total utility earned value. Instead, it evaluates to any number of total utility earned values depending on the sampled execution time value for each task. Maximizing utility at X^{th} percentile evaluates the given resource allocation N times, producing a unique total utility earned value for each evaluation. All N utility values are then sorted in descending order, and the value at the X^{th} percentile is selected. For example, if $N = 100$ and $X = 20$, the 20^{th} utility

value from the sorted list, u, would be selected for this resource allocation. This means that 20% of the time, this resource allocation would earn at least u utility. The values for N and X can be changed based on the needs of the system.

Minimizing Energy at X^{th} Percentile: The second probabilistic-based objective tries to minimize total energy consumed, and is similar to the probabilistic-based utility objective. A given resource allocation will evaluate to multiple total energy consumed values because we are using stochastic distributions for execution time and power consumption. Minimizing energy at X^{th} percentile evaluates the resource allocation N times and sorts the total energy consumed values in ascending order. The energy value at the X^{th} percentile is selected and returned. For example, if $N = 100$ and $X = 20$, the 20^{th} energy value from the sorted list, e, would be selected for this resource allocation. This means that 20% of the time, this resource allocation would consume less than or equal to e energy. The value for N is the same as the one used in the utility probabilistic-based objective, but the value for X may be different.

7.2.3. Generating Solutions for a Bi-Objective Optimization Problem.

When a problem contains multiple objectives, it is challenging to determine a single global optimal solution. Often, there is instead a *set* of optimal solutions. This set of optimal solutions may not be known, thus, it is important to find a set of solutions that are as close as possible to the optimal set. The set of solutions that we find that best approximates the optimal set is called the set of Pareto optimal solutions [72]. The Pareto optimal set can be used to construct a Pareto front that can illustrate the trade-offs between the two objectives.

Not all solutions we find can exist within the Pareto optimal set. We select the eligible solutions by examining solution dominance. Dominance is used to compare two solutions to

one another. For one solution to dominate another, it must be better than the other solution in at least one objective, and better than or equal in the other objective. Figure 7.2 illustrates the notion of solution dominance. In this figure, we show three potential solutions: A, B, and C. The objectives we are optimizing for are minimizing total energy consumption (shown along the x-axis) and maximizing total utility earned (shown along the y axis). A "good" solution would be one that consumes small amounts of energy and earns large amounts of utility (upper left corner).

If we consider solutions A and B, we can say that B is dominated by A. This is because A uses less energy as well as earns more utility than B. The same is true for any solution located within the lower right region. Now, when we examine solutions A and C, we cannot make any claims on the dominance of one solution over the other. This is because C uses less energy than A, but C does not earn as much utility, thus the two solutions can not be directly compared to each other, which means they are both located on the Pareto front. Finally, solution A will only be dominated by solutions that exist in the upper left region.

7.2.4. Nondominated Sorting Genetic Algorithm II. The Nondominated Sorting Genetic Algorithm II (NSGA-II) was designed for solving bi-objective optimization problems [61]. We enhanced a version of the NSGA-II that we used for the deterministic problem in [45] to handle stochastic execution times and power values. A description of the NSGA-II algorithm can be found in 5.4.4.

7.2.5. Pareto Fronts for the Different Objective Functions. For the expected value-based objectives, the values for total utility earned and total energy consumed that are used in the NSGA-II to rank solutions and also used to identify the location of the solution on the Pareto front are found by evaluating the resource allocation using expected

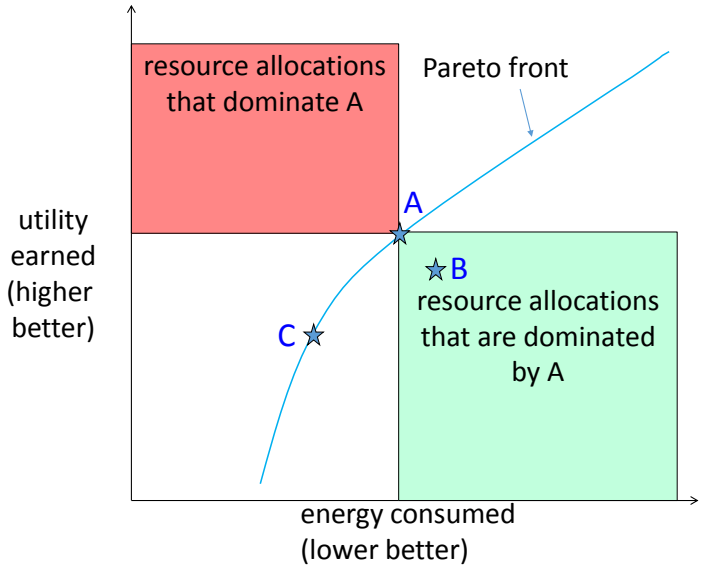


FIGURE 7.2. Illustration of solution dominance for three solutions: A, B, and C. Solution A dominates solution B because A has lower energy consumption as well as it earns more utility. Neither solution A nor C dominate each other because C uses less energy, while A earns more utility. A and C are a part of the Pareto front.

values for the task execution times. Conversely, these values, when using the probabilistic-based objectives, are found by identifying the X^{th} percentile total utility earned and total energy consumed values from the N evaluations using stochastic distributions for the task's execution times.

7.3. Simulation Setup

7.3.1. Workload Generation. A utility function for each task in a workload is given, and each task has a maximum utility value (depending on its priority level) that starts at one of 8, 4, 2, or 1. These values are based on the plans of the ESSC, but for other environments, different number of values and different values of maximum utility may be used. A method for generating utility functions for simulations can be found in Appendix F. Each task belongs to one among four urgency levels and 4 utility classes.

We simulate a system trace consisting of four hours worth of task arrivals (approximately 6000 tasks), where each task is one of 100 task types. Out of the 100 task types, 83 are general-purpose and 17 are special-purpose. Each task type has approximately the same number of tasks in it. We generate the arrival patterns to closely match patterns of interest to ESSC [2]. The general-purpose tasks arrive in a sinusoidal pattern and special-purpose tasks follow a bursty arrival pattern.

7.3.2. Execution Time and Power Modeling. The compute system that we model has 13 machine types (four are special-purpose) consisting of a total of 100 machines. The four special-purpose machine types have 2, 2, 3, and 3 machines in them. The remaining 90 machines are general-purpose and are split into the remaining nine machine types as follows: 5, 5, 5, 10, 10, 10, 10, 15, and 20. The machines of a special-purpose machine type run a subset of special-purpose task types approximately ten times faster, on average, than the general-purpose machines can run them (as discussed below). The special-purpose machines do not have the ability to run tasks of other task types. In our environment, three to five task types are special for each special-purpose machine type.

We assume that there exists three P-states in all machines: the highest power P-state (P-state 0), lowest power P-state, and an intermediate P-state. We use techniques from the Coefficient of Variation (COV) method [35] to generate the entries of the ETC and APC matrices in the highest power P-state. The mean value of execution time on the general-purpose and the special-purpose machine types is set to ten minutes and one minute, respectively. The mean dynamic power is set to 133 watts. To generate the dynamic power values for the intermediate P-state and the lowest power P-state, we scale the dynamic power to 75% and 50%, respectively, of the highest power P-state. The execution times for these P-states are also generated by scaling the execution time at the highest power P-state by sampling a

gamma distribution with a mean value that is approximately $1/\sqrt{(\%\ scaled\ in\ power)}$. For example, the lowest power P-state's execution time will be scaled by a value sampled from a gamma distribution that has a mean approximately equal to $1/\sqrt{0.5} = 1.414$. The execution time of any task is guaranteed to be the shortest in the highest power P-state, but the most energy-efficient P-state can vary across tasks (recall that the energy used is the product of the task's execution time and its power usage). Such a model approximates reality where the impact on execution time and energy consumption by switching P-states depends on, among other factors, the CPU-intensity/memory-intensity of the task.

To introduce stochasticity into this study, we create the probability distributions using gamma distributions. The mean value for these gamma distributions are set using the above-mentioned technique. The co-efficient of variation (COV) for these distributions is chosen uniformly at random from the range [0.1, 1.5]. This range creates a variety of distributions ranging from near-Guassian (at a COV of 0.1) to hyper-exponential (with a COV of 1.5), Figure 7.3 illustrates sample Gamma distributions at these values. After a distribution is created for a certain task-machine-P-state, we discretize the distribution to create its PMF that use for our study. In practical scenarios, one could use a variety of methodologies for modeling the stochastic behavior of task executions (e.g., [77–79]).

7.3.3. Experiments. We performed a variety of experiments to examine and analyze how using stochastic distributions for execution time and power consumption affect creating Pareto fronts. Each of the following experiments used the following parameters. The population size for the genetic algorithm is 100 chromosomes. Each population is allowed to run for 10,000 iterations. Tasks are not executed if they earn less than 0.5 utility.

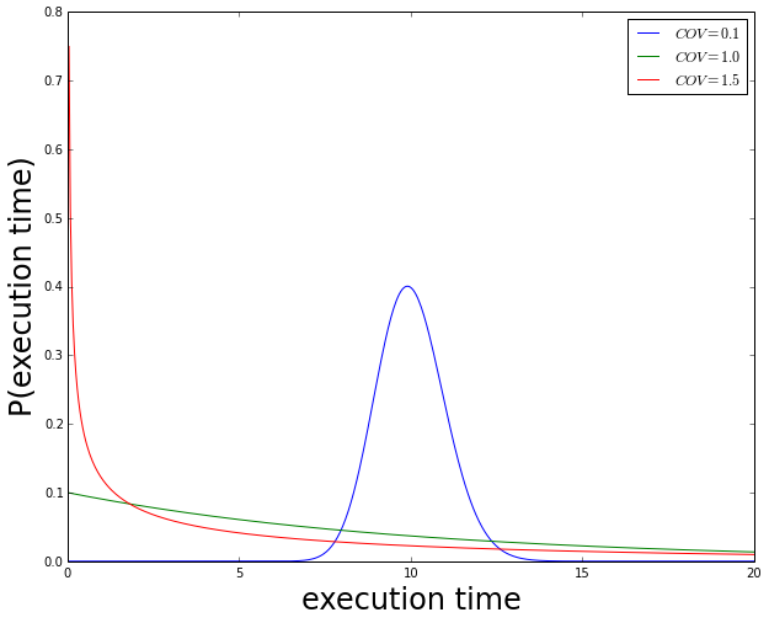


FIGURE 7.3. Gamma probability density functions with coefficient of variation (COV) values of 0.1 (near Gaussian), 1.0 (exponential), and 1.5 (hyper-exponential)

Our experiments compare the expected value-based objectives to the probabilistic-based objectives. For a given set of ETC and APC distributions, the expected value-based objectives will only evaluate to a single set of solutions, while the probabilistic-based objectives can be evaluated at different values of X^{th} percentile. We evaluated five different values of $X = 1$, 25, 50, 75, and 99. For each of the probabilistic-based objectives, $N = 100$.

7.3.4. GENETIC ALGORITHM SEEDS. Seeding heuristics provide the genetic algorithm with initial solutions that try to intelligently optimize one or both of the objectives. The seeds may help guide the genetic algorithm into better portions of the search space faster than an all-random initial population. We instrumented our dynamic simulator (described in this year's Task 1 and 2 reports) to be able to produce seeds from a post-mortem trace of any of the heuristics we have developed. The execution time required to run the dynamic simulator and produce seeds is negligible compared to the NSGA-II. Each seed is translated into a

chromosome within the genetic algorithms population, with the remaining chromosomes in the population being generated randomly. The dynamic simulator heuristics used to create seeds are described below.

Max-Max Utility is a two-stage greedy batch heuristic. In the first stage, the heuristic independently finds for each task in the batch the machine and P-state that maximizes the utility earned. In the second stage, the heuristic picks the task-machine-P-state choice from the first stage that provides the overall highest utility earned. The heuristic assigns the task to the machine and P-state, and iterates until all available tasks are mapped. **Max-Max Utility per Time** operates on the same principles as Max-Max Utility, except that the heuristic maximizes for "utility earned / execution time" instead of "utility earned." Similarly, **Max-Max Utility per Power** and **Max-Max Utility per Energy** optimize for "utility earned / average power consumption" and "utility earned / energy consumption," respectively.

7.4. Results

The results for the expected value-based objectives are shown in Figure 7.4. In this figure, each square marker represents an individual resource allocation. The utility and energy values for each square are found when evaluating the resource allocation using the expected value for each task's execution time. Recall, these are also the values calculated and used by the objective functions of the genetic algorithm. Associated with each square marker (resource allocation) are 100 circle markers. The utility and energy values for each circle are found by performing Monte Carlo evaluations of the resource allocation using stochastic distributions for each task's execution time. The combination of every resource allocation's circle markers form a "fuzzy" Pareto front. The square markers are not located

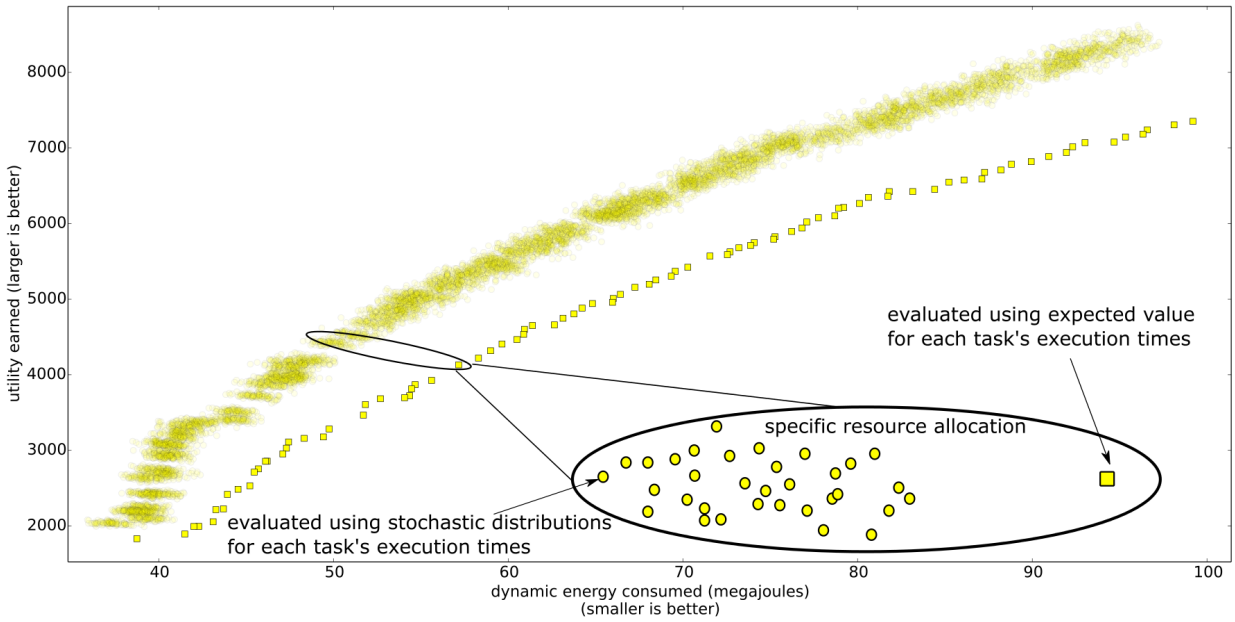


FIGURE 7.4. Expected value-based objectives results. Each square marker represents an individual resource allocation and the utility and energy values are found by evaluating the resource allocation using the expected value for each task's execution time. Each square marker (resource allocation) has 100 corresponding circle markers where the utility and energy values are found by performing Monte Carlo evaluations for the resource allocation using the stochastic distributions for each task's execution time.

in the middle of the fuzzy Pareto front due to the Gamma distributions used to model task execution times. For a Gamma distribution, median values are typically lower than the expected values. This indicates a task will evaluate to a faster execution time (than the expected value) more often than to a slower execution time. Additionally, for Gamma distributions with COVs in the range we are considering (0.1 - 1.5), the magnitude of the difference between the slower execution times and the expected value are larger than those between the faster execution times and the expected value. Thus, when evaluating every task for a given resource allocation, more tasks will evaluate to execution times faster than their expected values than tasks that evaluated to slower execution times, resulting in more utility earned and less energy consumed.

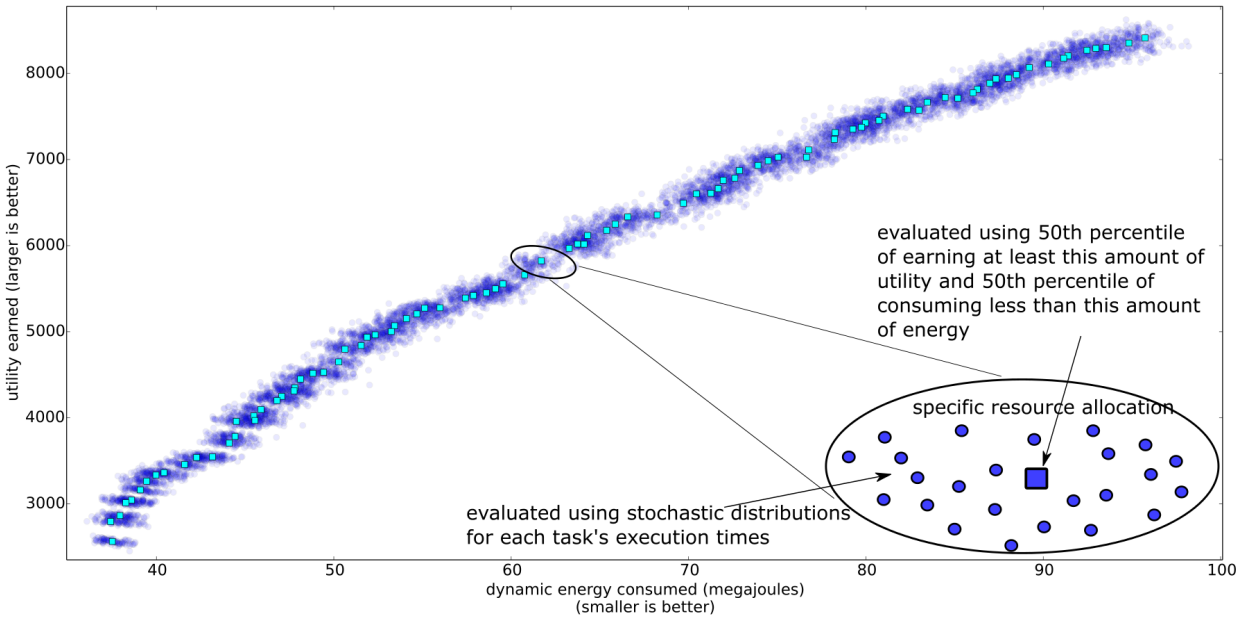


FIGURE 7.5. Probabilistic-based objectives with $X = 50\%$ results. Each square marker represents an individual resource allocation and the utility and energy values are found by evaluating the 50^{th} percentile of earning at least u utility and the 50^{th} percentile of consuming less than e energy. Each square marker (resource allocation) has 100 corresponding circle markers where the utility and energy values are found by Monte Carlo evaluations for the resource allocation using the stochastic distributions for each task's execution time.

The results for the probabilistic-based objectives with $X = 50$ are shown in Figure 7.5. In this figure, each square marker represents an individual resource allocation. The utility and energy values for each square are found when evaluating the resource allocation at the 50^{th} percentile of earning at least u utility, and at the 50^{th} percentile of consuming less than e energy. Recall, these are the values calculated and used by the objective functions of the genetic algorithm. Associated with each square marker (resource allocation) are 100 circle markers. The utility and energy values for each circle are found by performing Monte Carlo evaluations of the resource allocation using stochastic distributions for each task's execution time. For the probabilistic-based objectives, the square markers *must* lie within the fuzzy Pareto front because their utility and energy values are derived from the 100 evaluations of the resource allocations (circle markers). The results for $X = 1$, 25, 75, and 99 follow the

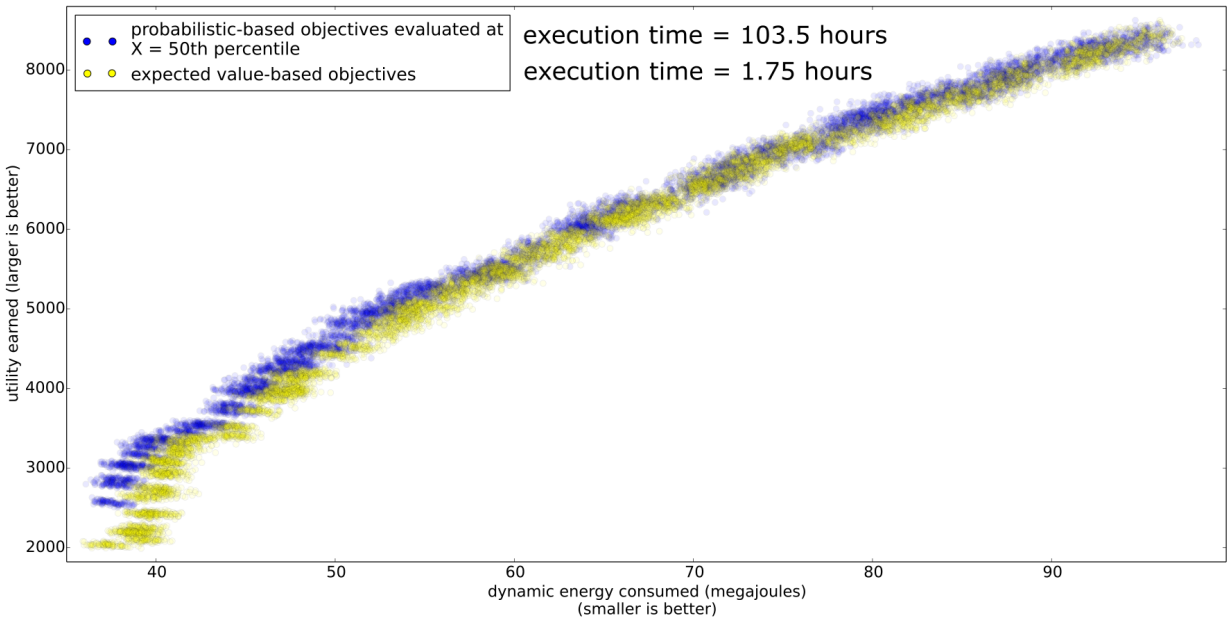


FIGURE 7.6. Comparison of fuzzy Pareto fronts for probabilistic-based objectives with $X = 50\%$ (blue markers) versus expected value-based objectives (yellow markers).

same behavior, but the location of the square marker changes from the upper left portion (less energy, more utility) ,when $X = 1$, to the lower right portion (more energy, less utility), when $X = 99$, of the circle markers for an individual resource allocation.

In Figure 7.6, we compare the fuzzy Pareto fronts of the probabilistic-based objectives with $X = 50$ (blue markers) to the expected value-based objectives (yellow markers). The fronts overlap significantly in the upper right portion of the figure (more utility, more energy), but the probabilistic-based objectives front dominates the expected value-based objectives front in the lower left portion of the figure (less utility, less energy). It is important to note that while the corresponding resource allocations of these fuzzy fronts are generated after the same number of NSGA-II iterations (10,000), the probabilistic-based objectives took over 100 hours while the expected value objectives took under 2 hours. The probabilistic-based objectives take significantly longer to execute because to calculate the X^{th} percentile values, N evaluations of the resource allocation must be performed. On the contrary, the expected

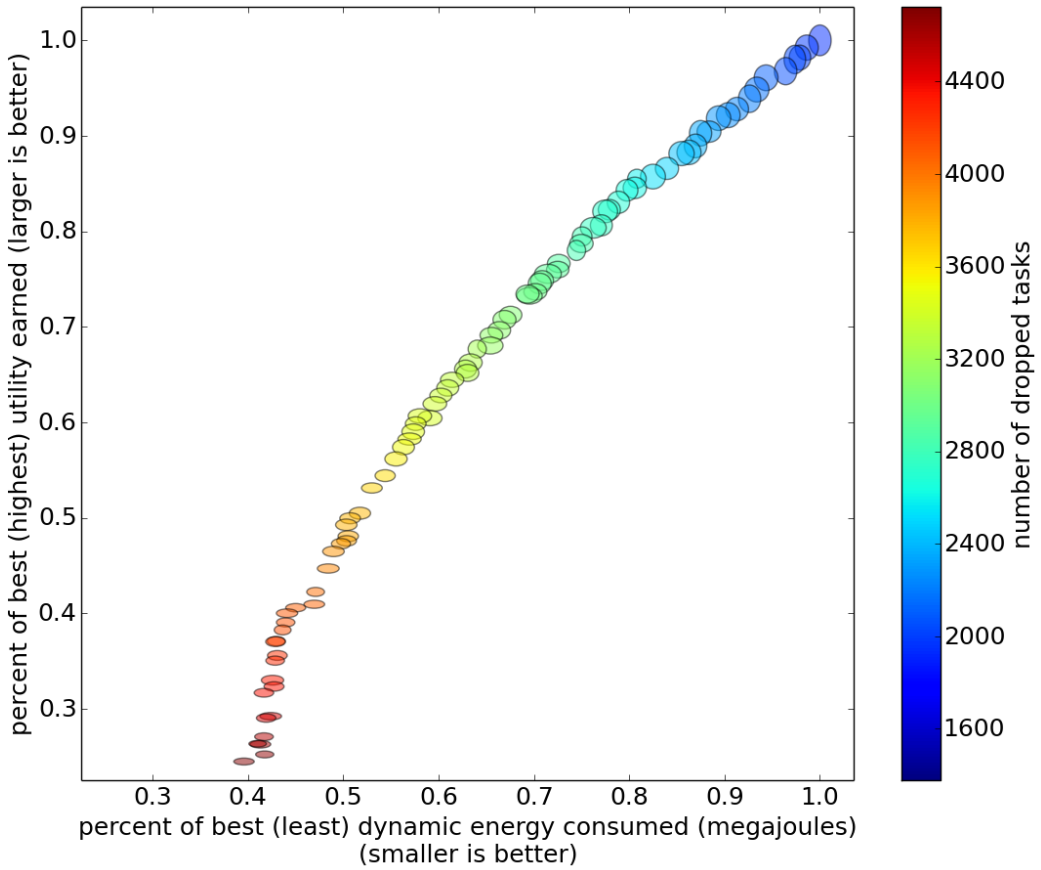


FIGURE 7.7. The variance and number of dropped tasks for resource allocations produced by the expected value-based objectives. The x-axis is normalized by the smallest value for energy consumed. The y-axis is normalized by the largest value for utility earned. The axes have a one-to-one aspect ratio. Smaller ellipses indicate less variance in the utility and energy for a given resource allocation. Vertical ellipses indicate more variance in utility than energy, while horizontal ellipses indicate more variance in energy than utility. Resource allocations that drop more tasks have less variation.

value-based objectives only perform a single evaluation of the resource allocation. A more fair comparison would be to let the expected value-based objectives version run for the same length of time as the probabilistic-based objectives version. These trends and analyses hold for the other probabilistic-based objectives experiments.

Fuzzy Pareto fronts can be used to analyze the utility and energy region that a specific resource allocation will evaluate to, and thus can provide a system administrator with insight into how much their system may vary for a given schedule, i.e. it provides more information than using just an expected value or an X^{th} percentile. The density and shape of a fuzzy Pareto front can indicate how much variance is present within the resources allocations corresponding to the front. To better visualize how the variance of resource allocations change across the front, Figure 7.7 shows each resource allocation, for the expected value-based objective fuzzy Pareto front, represented as an ellipse. The x-axis is normalized by the smallest value for energy consumed. The y-axis is normalized by the largest value for utility earned. The axes have a one-to-one aspect ratio. The size and shape of the ellipses indicate how much a given resource allocation varies with respect to energy and utility. The smaller the ellipse, the smaller the variance. Vertical ellipses indicate more variance with respect to utility than to energy, and horizontal ellipses indicate more variance with respect to energy than to utility. The color of each ellipse represents the number of tasks dropped by that given resource allocation, with blue meaning fewer tasks are dropped and red meaning more tasks are dropped. Resource allocations that drop a high number tasks earn less utility and consume less energy while also having less variation with respect to those values. This is because with fewer tasks, there is less uncertainty in the resource allocation, with respect to execution time, resulting in more consistent evaluations. We can also see that as fewer task are dropped, the variation in utility increases much more drastically than the variation in energy. This occurs due to the uncertainty in the task's execution times. Although energy and utility are both functions of execution time, energy increases linearly with execution time, while utility typically decreases non-linearly with execution time (depending on the shape of the utility function). In general, this causes utility to be much more sensitive

to changes in execution time than energy. The probabilistic-based objectives experiments exhibit similar trends.

7.5. Conclusions

Minimizing energy and maximizing utility are two important objectives for high performance heterogeneous computing systems. Unfortunately, these two objectives conflict with one another resulting in the need to analyze the trade-offs between them. The previous framework (based on the NSGA-II) presented in Chapter 5 was extend to account for uncertainties in task execution time and power consumption. These uncertainties are represented as Gamma distributions.

Two different sets of objective function operators were created. The first set is based on using expected values to represent task execution times and energy consumption. The second set of objectives are probabilistic-based where a given resource allocation is sampled N times and then the values for utility that would be earned and energy thar would be consume at the X^{th} percentile are selected. Numerous experiments were conducted to compare the expected value-based objectives to the probabilistic-based objectives (with numerous values for X).

Due to the stochastic nature of the task execution times, each resource allocation produced by the NSGA-II can be evaluated numerous times to create unique utility and energy values. 100 Monte Carlo evaluations were performed for each resource allocation. The combination of all 100 Monte Carlo evaluations for each resource allocation produces a "fuzzy" Pareto front. It was found that after the same number of generations, the probabilistic-based objectives fronts out preformed the expected value-based front (in terms of utility and energy), but at the cost of being over 50 times slower in algorithm execution time.

Finally, the variance was plotted with respect to utility and energy for each resource allocation, and it was found that resource allocations that earned lower amounts of utility and consumed less energy had less variance than resource allocations that earned higher amounts on utility and consumed more energy. This behavior is correlated to the number of tasks a given resource allocations drops. More dropped tasks correlates to less variance, while fewer dropped tasks correlate to more variance. Possible directions for future research are presented in Chapter 9.

CHAPTER 8

Modeling the Effects on Power and Performance from Memory Interference of Co-located Applications in Multicore Systems[1]

8.1. Introduction

There is an ever present desire to increase the performance and capabilities of current high performance computing systems. Frequently, increased performance comes at the cost of increased power dissipation, which has become a major challenge in large scale computing systems. According to the 2012 DatacenterDynamics census [40], global datacenter power requirements in 2007 were 12 GW, but doubled to 24 GW in 2011. Then, in 2012 the power requirements grew by 63% to 38 GW [40]. There is thus a critical need to profile and characterize power dissipation in computing nodes, as a precursor to developing techniques that can minimize power dissipation.

The use of multiple cores in today's processing units is commonplace as parallel processing remains a popular technique for speeding up the execution time of workloads. In an ideal system, doubling the number of cores in a server doubles the performance as long as the workload is perfectly parallel. However, cores in today's processors typically share resources in some manner (such as last-level cache, DRAM, network, and storage), causing contention

[1]This work was done jointly with Ph.D. student Daniel Dauwe. The full list of co-authors is at [80]. This work was supported by a National Science Foundation Graduate Research Fellowship, and the National Science Foundation under grant numbers CNS-0905399, CCF-1252500, and CCF-1302693 and by the Colorado State University George T. Abell Endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research also used the CSU ISTeC Cray System supported by NSF Grant CNS-0923386.

for these resources and degrading performance, potentially resulting in larger amounts of energy being consumed.

Understanding the effects of resource contention caused by co-locating applications in multi-core processors is becoming increasingly important as the number of cores per processor continues to increase. Co-location occurs when more than one application is executing and sharing resources on a multi-core processor. Different applications have different resource requirements, and by studying the execution time and energy effects across several applications one can better understand the implications associated with co-locating applications on multi-core processors. A better understanding of these effects becomes critical to intelligently mitigating interference and improving performance and energy consumption.

This work examines the effects of memory interference on energy usage and application execution time (or application throughput). Memory interference is introduced by executing a number of applications across the cores of a given processor, resulting in the applications sharing L2 or L3 caches in the memory hierarchy. Specifically, the research provides a testing infrastructure that monitors and collects data on the following attributes as various applications are executed on a multi-core processor: energy usage, execution time, and main memory (DRAM) accesses. Applications that access main memory frequently can be considered to be "memory intensive." This research also analyzes how applications with varying levels of memory intensity affect the attributes of applications with which they are co-located.

The attributes of four different workloads taken from the PARSEC benchmarking suite [81] are considered for measurement and analysis. The four workloads were selected to represent a range of applications with varying memory intensity. The workloads are executed on a quad-core Intel i7 processor. These experiments were chosen to be conducted on this

processor because it is being currently used in compute nodes within several datacenters that are increasingly deploying systems with low-end commodity processors (instead of high end server processors) to keep hardware costs manageable at a large scale. Results are first gathered for the case where each application executes by itself on a single core in the processor. These results are then used as a baseline to contrast the degradation in performance when there are two and four applications co-located on the cores of the processor. It is shown that for more memory intensive applications, there is a greater decrease in performance as memory interference increases. Additionally, this research incorporates the dynamic voltage and frequency scaling (DVFS) property of the processor, and runs all the tests across a range of different voltages and frequencies. The results from this research can be used to provide highly accurate execution time and energy consumption information for use in the area of resource allocation in high performance computing systems, where application tasks are co-located on the cores of multi-core processors[50, 45, 82].

In summary, this work makes the following contributions: (a) proposes a portable benchmark testing environment capable of measuring and analyzing the effects of cross-application interference on energy consumption, execution time, and memory accesses of various applications across CPU frequencies; (b) provides an analysis for a set of real-world workload execution scenarios using this testing environment to perform interference tests on a modern quad-core machine; and (c) gives insights into how large computer systems based on multi-core processors may be able to improve their energy use based on the memory interference caused by co-located applications.

The remainder of this chapter is organized as follows. The next section will discuss related work. Section 8.3 will describe the portable testing environment. The workloads being tested and the experimental setup will be explained in Section 8.4. Section 8.5 will

present and provide analyses for the results of the experiments. Conclusions and plans for future work will be discussed in Section 8.6.

8.2. Related Work

The authors from [83] perform experiments to measure the effect of varying processor frequency on the energy consumption of workloads with varying levels of memory intensity on a single system. Multiple instances of each application are executed concurrently on the test system with each instance pinned to a separate core. The results of [83] imply that the memory intensive tasks may be more energy efficient at slower frequencies while CPU intensive tasks may be more efficient at higher frequencies. The work in [83] does not quantify memory intensity nor does it consider the effects of co-locating applications, be it instances of the same application or instances of different applications.

A study of how different architectures can affect the impact of DVFS on energy savings is analyzed in [84]. Three generations of AMD processors from 2003 through 2009 are tested using a memory intensive workload across the various frequencies available to the processors. It was found that for the older processors, the most energy efficient frequencies existed in the middle of the dynamic frequency range, while for the newest processor, it was most energy efficient to run at the fastest frequency. The authors of [84] only examine a single application and do not consider how performance may be affected by co-location.

The work in [85] presents a study that investigates the impact of co-locating threads from multiple applications with diverse memory behavior on a quad-core system. The authors show that the execution time of co-located tasks can vary greatly depending on the other tasks that are executing. The impact co-location has on energy consumption is not considered

in [85], nor does the work try to isolate the effects different applications have on each other by pinning applications to specific cores.

The memory characterization of workloads from SPEC CPU2000 and SPEC CPU2006 are analyzed and presented in [86]. It is shown that changes in the size of the cache of a processor can greatly affect the memory intensity of a given workload. This implies that as cache size increases, a workload can switch from being memory intensive to being CPU intensive. It is important to study the effects co-location could have on such tasks as it may make them memory intensive again once they have to contend for cache with other applications. The work in [86] does not consider co-locating nor does it consider energy consumption.

A method is presented in [87] to predict the performance degradation of workloads from interference due to shared processor cache (co-location). Each workload is encapsulated within its own virtual machine, and each virtual machine is pinned to its own core. The authors were able to reasonably predict the degradation due to co-location for various applications. The work in [87] does not consider how the frequency of the CPU can affect the performance of the workloads nor does it examine impact of co-location on the energy consumption of individual tasks.

8.3. Testing Environment

8.3.1. Operating System. One of the goals of this research is to create a testing environment that is portable across a variety of machines and system architectures, as well as being "lightweight" to minimize noise that may occur from OS (operating system) jitter. OS jitter occurs when processes unrelated to the tested workload are executed (i.e., graphics procedures, mail daemons, security services, etc.). These processes can cause anomalies to

appear in the data. The operating system used for this research is a lightweight command-line version of Ubuntu 12.04 Linux. The operating system runs Linux kernel version 3.8.0.29-generic and uses the Linux default "SCHED_OTHER" thread scheduling policy.

To ensure consistency across different experiments, any power saving features of the operating system have been disabled (e.g., screensaver and automatic processor throttling). The operating system has been configured in such a way that it can be used on a variety of different systems. Currently, it has been installed onto a bootable USB drive, but in the future, the OS could be run from a live CD or from a netboot.

8.3.2. Processor Performance Counters. Processors today have the ability to measure and report on numerous hardware events such as the number of cache misses or number of instructions executed through the use of performance counters. By recording different events, it is possible to gain insight into the characteristics of a given application on a given processor architecture. Typically, there are a large number of hardware events that can be measured but a small number of performance counters (e.g., in an Intel i7 there are only seven performance counters, but over 50 different measurable events [88]) meaning that only a limited number of events can be measured at any given time. To further complicate matters, due to differences between microarchitectures, the number, type, and availability of performance counters and measurable events can vary greatly from system to system. To allow the testing environment to be portable, additional tools are used to monitor and collect performance counter data across multiple systems.

The first tool used is the Performance Application Programming Interface (PAPI) which, is a portable API to hardware performance counters that simplifies interfacing between software and the native hardware performance counters of processors [89]. Due to the issues involved with the variability in numbers and types of native hardware events available across

microprocessor architectures, PAPI attempts to define a standard set of performance counters ("presets") that can be found in most microprocessors. These preset performance counters try to abstract away the architecture specific details to provide an easy way to manipulate and measure similar hardware events across numerous platforms. The PAPI library contains over 100 preset performance counters that are available for use [90], however as stated earlier due to differences in architectures, not all performance counters are available on every system.

The HPCtoolkit [91] interfaces with PAPI and provides a set of software tools that facilitates measuring and collecting the performance data of an application. The HPCtoolkit was designed to perform benchmark testing using performance counters and therefore there is very little overhead associated with utilizing this tool. Specifically for this research, the "hpcrun-flat" tool was used to execute and monitor the test applications.

8.3.3. Measuring Memory Intensity. To understand the effects of co-locating multiple applications, additional metrics of performance besides execution time and energy consumption should be measured. One hypothesis of this research is that applications that frequently need to access the last-level of cache and main memory have a greater degradation in performance when co-located with other memory intensive applications compared to applications that are mostly CPU intensive. To test this hypothesis, the measure used here for relative memory intensity is last-level cache misses divided by instructions executed.

Relative memory intensity indicates how often an application must access main memory (DRAM) per instruction executed. The number of times an application accesses main memory can be measured by keeping track of the number of last-level cache misses. The total number of last-level cache misses is, however, not enough to classify the memory intensity of an application, as different applications can execute for different amounts of time, and have different instruction counts. Therefore, the last-level cache misses are normalized by

the number of total instructions executed. The resulting metric called relative memory intensity allows for the comparison of memory intensity across applications regardless of the instruction counts of the applications.

The number of last-level cache misses and the total number of instructions executed are measured using performance counters. PAPI does not contain a standard "last-level cache miss" performance counter and thus the appropriate level (L2 or L3) cache miss event must be set depending on the specific microprocessor architecture.

8.3.4. Processor Performance States (P-states). Performance states (P-states) utilize dynamic voltage and frequency (DVFS) capability in processors to control the power consumption and speed at which the processor is operating. P-states are denoted by integer numbers, with P-state P0 indicating the highest voltage and highest (fastest) frequency. Higher P-state numbers indicate lower voltages and lower (slower) CPU frequencies. The number of P-states available for any given CPU, as well as the voltage and frequency pairings that each P-state represents, are different for every processor architecture, and therefore trends and behaviors of applications on one system may be significantly different when compared to another system. Generally, across different processor microarchitectures, operating in higher P-states results in increased execution time.

Furthermore, P-states only control the portion of a processors total power consumption called the dynamic power. The remaining power in the processor is called the static power and is assumed to remain constant regardless of the P-state. The ratio of dynamic power to static power differs from architecture to architecture, and using P-states to decrease the CPU frequency and voltage will decrease the dynamic power use but also likely increases an application's execution time. Due to static power being constant across P-states and

increased execution time, there is no guarantee that operating in a slower P-state will result in reduced energy consumption.

This research provides an infrastructure for analyzing how the performance in terms of execution time, energy consumption, and memory intensity changes across different P-states, in modern multi-core processor architectures.

8.3.5. "Watts Up? PRO" Power Meter. A *Watts Up? PRO* power meter [92] was used to collect and calculate the energy used while an application executes on microprocessors. The *Watts Up?* meter connects to and measures the system at the "outlet" level, meaning that all the power used by the whole system is measured. The meter is able to measure the instantaneous power draw of the system at one second intervals. To calculate the energy consumed by an application, the power samples, which are the average power over the one-second intervals and so are simply summed over the execution time.

8.4. Experimental Setup

8.4.1. PARSEC Benchmark Suite. The workloads used in this research are taken from the PARSEC benchmarking suite [81], which consists of a diverse set of applications from many different computing areas. Four benchmarking applications were chosen from the PARSEC suite with the intention of providing a representative set of applications having varying degrees of memory intensity. The applications chosen for experimentation, along with a brief description, can be found in Table 8.1, organized from most memory intensive to least memory intensive.

8.4.2. Co-location Experiments. To measure the effect of co-location on the energy consumption, execution time, and relative memory intensity of an application, a set of baseline tests were conducted to establish measurements that these three metrics could

Table 8.1. PARSEC Applications

Application	Description
canneal	cache-aware simulated annealing to optimize routing cost of a chip design
streamcluster	online clustering of an input stream
blackscholes	option pricing with Black-Scholes Partial Differential Equation (PDE)
bodytrack	body tracking of a person

be compared to, across changes in the number of processor cores. For a baseline test, the applications were executed by themselves on the multi-core processor. During execution, the application was pinned to a specific core using the Linux "taskset" command and was the only process executing other than OS-related processes. Additionally, each baseline test was executed at different processor frequencies (P-states) ranging from 3.40GHz, the CPUs default highest speed, down to 1.70GHz. Energy consumption, execution time, and relative memory intensity were measured for each application in each P-state.

From the relative memory intensity results of the baseline tests (Table 8.2), the applications were then classified into three groups based on the magnitude of their relative memory intensities. The applications are shown in order of decreasing memory intensity. As indicated in the table, canneal is the most memory intensive application with over two last-level cache misses per 100 instructions executed while blackscholes is the least memory intensive application with fewer than one last-level cache miss per 100,000 instructions executed.

Table 8.2. Memory Intensity Classification

Applications	Classification	Relative Memory Intensity
canneal	Intensity III	2.25×10^{-2}
streamcluster	Intensity III	1.64×10^{-2}
blackscholes	Intensity II	2.29×10^{-5}
bodytrack	Intensity I	7.44×10^{-6}

The canneal and streamcluster applications exhibit the most interactions with main memory, and are categorized as being "Intensity III" memory intensive tests, blackscholes is categorized a "Intensity II" memory intensive application, and the bodytrack application, accessing memory the least, is categorized as being a "Intensity I" memory intensive application.

To test the effect of co-location on energy use, execution time, and relative memory intensity, two additional sets of experiments were performed. The first set contains experiments where two applications were co-located each application is pinned to its own core. Each application could be paired with one of the other applications or a copy of itself. For brevity, a subset of these possible pairs is presented in Table 8.3. The selection of this subset of pairs is due to the fact the canneal application is the most memory intensive application that was tested. canneal is therefore tested against the other applications to determine how those applications are impacted when co-running with a highly memory intensive application.

The second set of experiments increases the level of co-location from two applications up to four applications, using the "taskset" command to pin each application to its own core within the quad-core processor. Again, a subset of the numerous possible combinations for co-locating the applications is presented in this work for brevity. These subset combinations are shown in Table 8.4.

TABLE 8.3. Two Core Interference Tests

Test Type	Applications co-located together
2 Intensity III	canneal, streamcluster
1 Intensity III and 1 Intensity II	canneal, blackscholes
1 Intensity III and 1 Intensity I	canneal, bodytrack

Table 8.4. Four Core Interference Tests

Test Type	Applications co-located together
4 Intensity III	2 canneal, 2 streamcluster
1 Intensity III, 1 Intensity II, and 2 Intensity I	1 canneal, 1 blackscholes, 2 bodytrack
2 Intensity III and 2 Intensity I	1 canneal, 1 streamcluster, 2 bodytrack

8.5. Results

For the results presented in this section, tests were performed on the 64-bit Intel i7 3770 3.40 GHz quad-core processor [93]. In this processor, each core has its own private L1 and L2 caches, but a shared (8MB) L3 cache, that is shared by all four cores. The operating frequency of the processor can vary from 3.40 GHz to 1.70 GHz over the range of P-states, and because DVFS is used to do this, it prevents the processor from over-clocking the CPU into Intel's "turbo" mode during the baseline tests when only one core is being used. The average of nine runs of each test are reported.

The results of the co-location tests are shown in Figures 8.1 to 8.3. Each figure contains the results for relative memory intensity, execution time, and energy consumption of a single application across multiple processor frequencies for the baseline (red), two-way co-location (green), and four-way co-location (blue) tests. Subfigure (a) of each figure shows the relative memory intensity, Subfigure (b) shows the execution time, and Subfigure (c) shows the energy consumption.

Note that for the memory intensity and execution time subfigures (Subfigures 8.1(a) and (b) through Subfigures 8.4(a) and (b), the results shown are for the individual application being presented. For example, when looking at the execution times of canneal in Subfigure

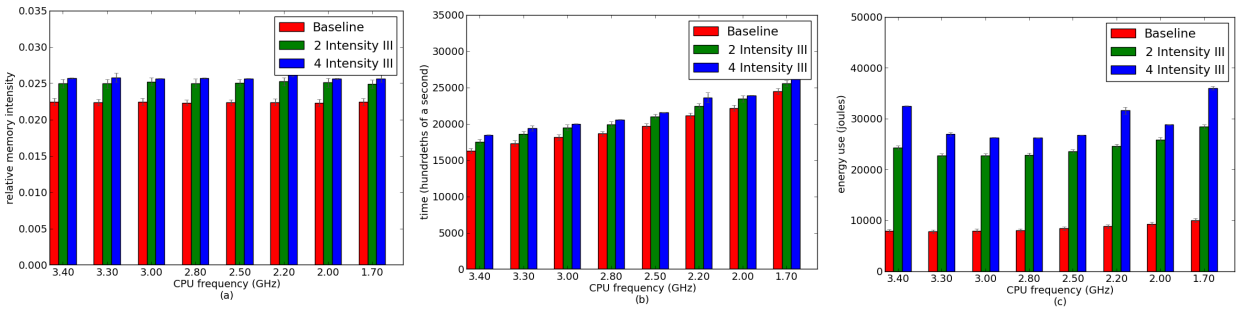


FIGURE 8.1. The baseline (red), two-way co-location (green), and four-way co-location(blue) tests for canneal. The x-axis of each subfigure is the frequency of the processor. (a) Memory intensity results, the y-axis is the relative memory intensity measured as the number of last-level cache misses per instruction executed. (b) Exectuion time results, the y-axis is the execution time measured in hundredths of a second. (c) Energy consumption results, the y-axis is the energy consumed measured in Joules.

8.1(b), the execution times for "2 Intensity III" and "4 Intensity III" bars are the actual execution times of only the canneal application, not the execution times of all the co-located applications. In contrast, due to the fact the *Watts up? PRO* meter takes measurements at the "outlet" level, the energy results in Subfigures 8.1(c) through 8.4(c) show the energy consumption for all co-located applications. Furthermore, because the applications have different execution times, the energy consumed that is reported for an application is the total energy consumed during that application's execution time.

Figures 8.1 and 8.2 contain the results for the canneal and streamcluster applications. Recall that canneal and streamcluster are the two "Intensity III" memory intensive applications that were tested. When examining the effect of co-location on the relative memory intensity Subfigures 8.1(a) and 8.2(a) it is apparent that when multiple Intensity III applications are co-located the number of last-level cache misses increase, resulting in more memory accesses, and implying that the applications are creating interference by evicting lines from the last-level cache that were being used by other applications. The large variations between

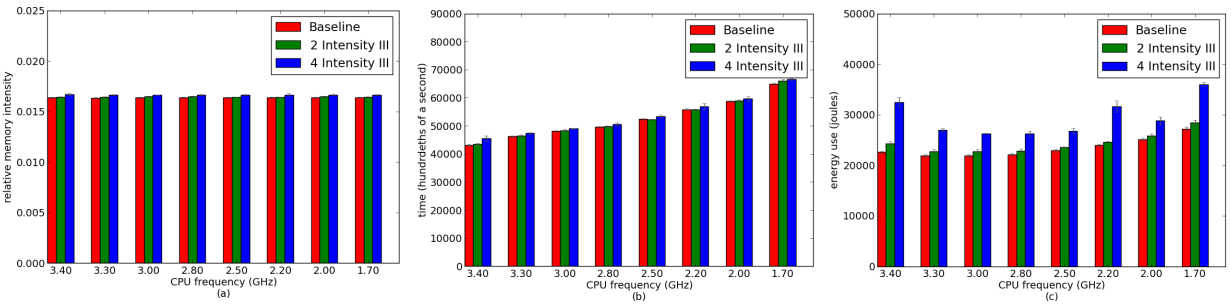


FIGURE 8.2. The baseline (red), two-way co-location (green), and four-way co-location(blue) tests for streamcluster. The x-axis of each subfigure is the frequency of the processor. (a) Memory intensity results, the y-axis is the relative memory intensity measured as the number of last-level cache misses per instruction executed. (b) Exectuion time results, the y-axis is the execution time measured in hundredths of a second. (c) Energy consumption results, the y-axis is the energy consumed measured in Joules.

the baseline results and the dual and quad-core interference results seen between every application is a result of differences in how memory interference affects each application. It should be noted that the scale of each graph changes between figures. The execution time of both applications are shown in Subfigure 8.1(b) and Subfigure 8.2(b) respectively. Co-location increases the execution time of applications.

The results for the blackscholes and bodytrack applications are shown in Figures 8.3 and 8.4, respectively. These two application are significantly less memory intensive than canneal and streamcluster. The results for relative memory intensity (Subfigures 8.3(a) and 8.4(a)) indicate that co-location can greatly increase the memory intensity of "Intensity I" memory intensive tasks in a relative sense, but it is important to note that even in the worst case (highest memory intensity) these two applications still have fewer last-level cache misses than either of the best cases (lowest memory intensity) for the canneal and streamcluster applications. This is also the reason for the apparent fluctuations in the memory intensity values for blackscholes and bodytrack. As can be seen by examining the variations shown in the error bars, even the smallest variations present in the highly memory intensive canneal

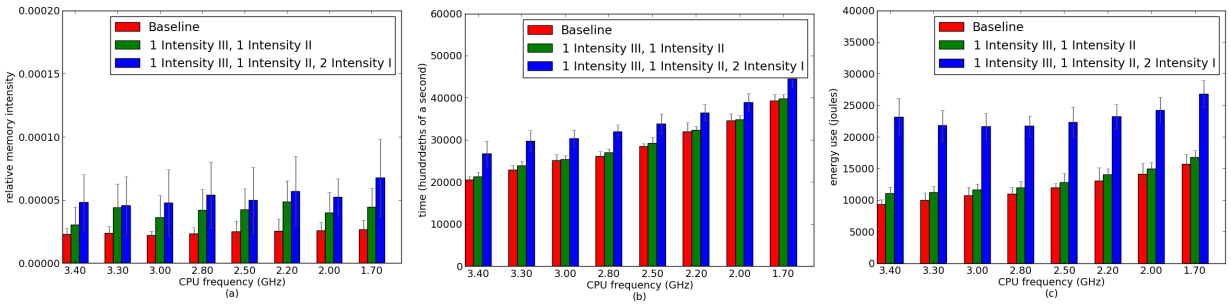


FIGURE 8.3. The baseline (red), two-way co-location (green), and four-way co-location(blue) tests for blackscholes. The x-axis of each subfigure is the frequency of the processor. (a) Memory intensity results, the y-axis is the relative memory intensity measured as the number of last-level cache misses per instruction executed. (b) Execution time results, the y-axis is the execution time measured in hundredths of a second. (c) Energy consumption results, the y-axis is the energy consumed measured in Joules.

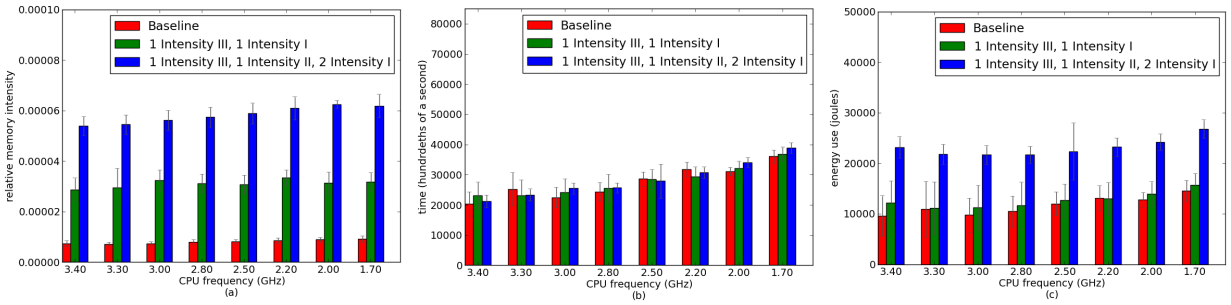


FIGURE 8.4. The baseline (red), two-way co-location (green), and four-way co-location(blue) tests for bodytrack. The x-axis of each subfigure is the frequency of the processor. (a) Memory intensity results, the y-axis is the relative memory intensity measured as the number of last-level cache misses per instruction executed. (b) Exectuion time results, the y-axis is the execution time measured in hundredths of a second. (c) Energy consumption results, the y-axis is the energy consumed measured in Joules.

data is still greater than the largest variations in the blackscholes data (a difference of 7.05×10^{-5} for canneal as compared to 6.16×10^{-5} for blackscholes). Further analysis of the execution times for blackscholes and bodytrack (Subfigures 8.3(b) and 8.4(b)) indicates that even though co-location may have a significant relative increase in memory intensity for these applications, it minimally affects the execution times.

In general, across all applications, changing processor frequency (P-state) has minimal impact on the relative memory intensity (Subfigures 8.1(a), 8.2(a), 8.3(a), 8.4(a)), as processor speed should not affect the memory behavior of an application. Changing processor frequency does have a noticeable impact on execution time (Subfigures 8.1(b), 8.2(b), 8.3(b), 8.4(b)). As expected, when the frequency decreased, the execution times of all the applications increased. This increase in execution time is not uniform across all the applications. In general, the "Intensity I" memory intensive tasks are more sensitive to changes in frequency, meaning they experience a larger percent increase in execution time compared to the "Intensity III" memory intensive tasks. Most likely this is due to the fact that the "Intensity I" memory intensive tasks perform most of their computing on the CPU, operating mainly out of the cache, thus they do not need to access main memory very often.

An important distinction to make when analyzing the energy results (Subfigures 8.1(c), 8.2(c), 8.3(c), 8.4(c)) of these experiments is that the baseline results show the energy consumed during a single task's execution, while the co-location results show the energy consumed for *multiple* task's execution. For example in Figure 8.1(c) the energy use shown in the graph is a measure of the entire test's energy use up until the time that the canneal application finishes executing, whereas while the energy use of streamcluster in Figure 8.2(c) is taken from the same data measurements as that of Figure 8.1(c) the execution time of the the streamcluster application is longer, so its power use is greater than that of canneal.

It can be observed that even though the energy consumed to execute a task increases when co-located with other tasks, the total energy used to completely execute all the tasks actually decreases. For example, as detailed in table 5, if canneal and streamcluster were executed independently without any co-location they would require approximately 30,509 Joules total to run, but when co-located with one another they require approximately only 24,281 Joules

total. This is because when run independently, both applications will separately incur the static energy present in the system in addition to the dynamic energy used during their execution. When co-located, both applications are able to share the static power resulting in less static energy being consumed during execution, and less total energy used overall. Only a subset of these results are shown in the table, but they are consistent across all test runs. Furthermore, from the Subfigures 8.1(c), 8.2(c), 8.3(c), 8.4(c) it can be seen that the optimal p-state for minimizing dynamic energy is different between applications, and can change between an application's baseline, two-core, and four-core co-location tests.

This behavior along with the fact that minimal performance degradation occurs when "Intensity III" and "Intensity I" memory intensive tasks are co-located with one another provides many interesting and exciting possibilities for smart resource allocation managers in high performance computing (HPC) systems. For example, task schedulers could use this information to intelligently co-locate applications with differing memory intensities on multi-core processors in server nodes to minimize performance loss and decrease system energy consumption.

TABLE 8.5. Test Energy Savings from Sharing Static Power (units in Joules, results are taken from tests run at 3.40GHz)

Application	Energy Use
canneal	7873
streamcluster	22,636
blackscholes	9347
bodytrack	9583
canneal + blackscholes	11,055 (Co-located)
canneal + streamcluster	24,281 (Co-located)
canneal + blackscholes + 2 bodytrack	28,729 (Co-located)
2 canneal + 2 streamcluster	32,504 (Co-located)

8.6. Conclusion and Future Work

With the desire for increased performance in computing systems, multi-core processors have become a popular and prevalent method of achieving higher performance. These multi-core processors are able to execute multiple applications at one time, however there exist complex interactions between the memory access behavior of applications that may cause degradations in performance and increased energy consumption for each individual application. This work examined the impact of memory interference on execution time, energy consumption, and relative memory intensity for co-located applications on multi-core processors. Specifically, a portable and lightweight testing framework was presented where four workloads taken from the PARSEC benchmark suite were run on an Intel i7 quad-core machine. The results verify that applications that have "Intensity III" memory intensity are more sensitive in terms of execution time and energy consumption when co-located with other "Intensity III" memory intensive applications, while "Intensity I" memory intensive applications are much less susceptible to degradations in their performance when co-located with other applications. For the specific system tested, it was shown that the optimal processor frequency for minimizing energy consumption could change based on the application and the co-location workload due to variations in use of dynamic power, the increased execution time, and static power present within the system. It was also found that co-running applications can also lead to sharing of static power use among the simultaneously running applications, which ends up decreasing the energy consumed for each application. Changing the frequency of the processor had negligible effect on the memory intensity of the applications. Possible directions for future research are presented in Chapter 9.

CHAPTER 9

Future Work

The study in Chapter 2 presented a flexible metric that uses utility functions to compare the performance of resource allocation heuristics in an oversubscribed heterogeneous computing environment. Several utility maximizing heuristics were presented, and produced resource allocations that significantly outperformed, more classical approaches. Possible directions for future research extending this include: (a) using stochastic estimates of execution time to more closely model a real environment and to analyze the tolerance of the resource management policies to such uncertainties, (b) obtaining a model of expected arrival time of tasks from historical data so that the dropping threshold could be varied dynamically throughout the day based on the expected system load, (c) introducing utility functions that do not have to be monotonically-decreasing, (d) introducing parallel jobs (that require multiple machines concurrently to execute), (e) permitting pre-emption of tasks, (f) developing heuristics that take the utility-functions' slopes into consideration to guide their resource allocation decisions, and (g) adapting the heuristics to work with a semi-distributed resource manager [94].

Chapter 3 expanded on the study in Chapter 2 by introducing an energy constraint into the system. This required the design of new heuristics that were energy-aware. Additionally an energy filtering technique was created to add energy-awareness into non energy-aware heuristics and techniques. One of the main goals of this study was to perform an in-depth analysis of the performance of the energy-constrained heuristics. As part of future work, this knowledge of the performance of the heuristics, filtering, and weighting techniques can be used to design adaptive techniques that can auto-tune the value of *energy leniency* and/or

U-E weighting factor dynamically. Other possible directions for future research include: (a) designing energy-aware robust resource allocation techniques that account for various sources of uncertainty, such as stochastic task execution times, (b) creating different patterns for the arrival of utility into the system (e.g., high utility tasks arriving for certain fixed times of the day) and designing techniques that can account for and adapt to these changes, (c) designing heuristics that use the information about the slope of the utility-functions to make allocation decisions, (d) considering workloads of dependent and parallel tasks to broaden the scope of this work, (e) extending the model to simulate task and machine failures, and (f) exploring how to represent service level agreements using utility functions.

Chapter 4 continues the examination of dynamic heuristics in energy constrained environments. Specifically, this chapter examines how different task management environments (queued and polled) affects the performance of various heuristics. Possible directions for future research include: (a) making the resource allocation techniques robust to uncertainties such as stochastic task execution times, machine failures, etc., (b) experimenting with other types of task management environments, and (c) considering parallel and dependent tasks scheduling.

The study in Chapter 5 transitions from dynamic to static resource allocation by performing a post-mortem analysis on a system trace to analyze the trade-offs between system utility earned and system energy consumed. It was found that for well structured resource allocations, and increase in energy typically leads to an increase in performance. There are many possible directions of future work for this study including: (a) a technique to drop tasks that generate negligible utility when they complete could be implemented into the genetic algorithm to analyze the effect of dropping tasks on performance and energy consumption, (b) incorporating dynamic voltage and frequency scaling capabilities of processors would

allow for more freedom when making energy conscious decisions, (c) a comparison between the NSGA-II and other bi-objective algorithms, (d) to increase the performance of the algorithm, it could be adapted to use an "island model" and then be analyzed in detail to see how it affects performance in this type of environment.

The study performed in Chapter 6 examined the trade-offs between makespan and system energy consumed. A machine-by-machine analysis was performed to understand how resource allocations on individual machines change throughout solutions in the Pareto front. This study can be extended in many ways, including: (a) extending the power model to incorporate dynamic voltage and frequency scaling techniques, (b) increase the execution rate and performance of the genetic algorithm by trying numerous parallel techniques, (c) examine how the structure of the chromosome can affect to execution rate of the genetic algorithm, and (d) examine different performance and cost objectives such as: maximizing robustness, minimizing temperature, maximizing profit, and minimizing monetary costs.

Chapter 7 extends the bi-objective optimization framework presented in Chapters 5 and 6 by representing task execution times and energy consumption values scholastically as probability distributions. The stochastic nature of the problem results in multiple viable realizations of each resource allocation. By evaluating each resource allocation in the Pareto front numerous times a "Fuzzy" Pareto front is created. This study can be extended in many ways, including: (a) using real historical data of tasks and machines to represent the probability distributions used by the framework, (b) implement techniques similar to those found in stochastic programming to calculate the value of perfect information and value of the stochastic solution, and (c) designing techniques to try and minimize to variability of the resource allocations.

The effects of co-locating mutliple applications on the same multicore processor was examined in Chapter 8. It was shown that performance and energy consumption can be greatly influenced by the types of applications co-located with one another, allowing for some interesting possible directions for future work including: (a) increasing the number of PARSEC test applications as well as including applications from additional benchmark suites, (b) performing co-location tests on a variety of systems and architectures, (c)information gathered from these tests could then be used by high performance scheduling systems to co-locate applications in a manner that minimizes performance degradation and energy consumption, and(d) extended these techniques to resource management in heterogeneous multicore-based distributed systems; e.g., [67], [10], [95].

Bibliography

[1] D. Filani, J. He, S. Gao, M. Rajappa, A. Kumar, P. Shah, and R. Nagappan, "Dynamic data center power management: Trends, issues, and solutions," *Intel Technology Journal*, vol. 12, no. 1, pp. 59–67, Feb. 2008.

[2] B. Khemka, R. Friese, L. D. Briceño, H. J. Siegel, A. A. Maciejewski, G. A. Koenig, C. Groer, G. Okonski, M. M. Hilton, R. Rambharos, and S. Poole, "Utility functions and resource management in an oversubscribed heterogeneous computing environment," *IEEE Transactions on Computers*, 2014, accepted for publication.

[3] E. Jensen, C. Locke, and H. Tokuda, "A time-driven scheduling model for real-time systems," in *International Real-Time Systems Symposium*, Dec. 1985, pp. 112–122.

[4] B. Ravindran, E. D. Jensen, and P. Li, "On recent advances in time/utility function real-time scheduling and resource management," in *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005)*, May 2005, pp. 55–60.

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

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

[7] C. B. Lee and A. E. Snavely, "Precise and realistic utility functions for user-centric performance analysis of schedulers," in *International Symposium on High Performance Distributed Computing (HPDC '07)*, 2007, pp. 107–116.

[8] J.-K. Kim, H. J. Siegel, A. A. Maciejewski, and R. Eigenmann, "Dynamic resource management in energy constrained heterogeneous computing systems using voltage scaling,"

IEEE Transactions on Parallel and Distributed Systems, vol. 19, no. 11, pp. 1445–1457, Nov. 2008.

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

[10] 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 environments," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 326–347, Feb. 2013.

[11] H. Barada, S. M. Sait, and N. Baig, "Task matching and scheduling in heterogeneous systems using simulated evolution," in *10th Heterogeneous Computing Workshop (HCW 2001), in the proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, Apr. 2001, pp. 875–882.

[12] M. K. Dhodhi, I. Ahmad, and A. Yatama, "An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 62, no. 9, pp. 1338–1361, Sep. 2002.

[13] A. Ghafoor and J. Yang, "A distributed heterogeneous supercomputing management system," *IEEE Computer*, vol. 26, no. 6, pp. 78–86, June 1993.

[14] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *IEEE Concurrency*, vol. 6, no. 3, pp. 42–51, July 1998.

[15] A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C. Wang, "Heterogeneous computing: Challenges and opportunities," *IEEE Computer*, vol. 26, no. 6, pp. 18–27, June 1993.

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

[17] M. R. Gary and D. S. Johnson, *Computers and Intractability: A guide to the theory of NP-Completeness*. W. H. Freeman and Co., 1979.

[18] C. M. Krishna and K. G. Shin, *Real-Time Systems*. McGraw-Hill, 1997.

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

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

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

[22] P. Sugavanam, H. J. Siegel, A. A. Maciejewski, M. Oltikar, A. Mehta, R. Pichel, A. Horiuchi, V. Shestak, M. Al-Otaibi, Y. Krishnamurthy, S. Ali, J. Zhang, M. Aydin, P. Lee, K. Guru, M. Raskey, and A. Pippin, "Robust static allocation of resources for independent tasks under makespan and dollar cost constraints," *Journal of Parallel and Distributed Computing*, vol. 67, no. 4, pp. 400–416, Apr. 2007.

[23] I. Al-Azzoni and D. G. Down, "Linear programming-based affinity scheduling of independent tasks on heterogeneous computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 12, pp. 1671–1682, Dec. 2008.

[24] J.-K. Kim, S. Shivle, H. J. Siegel, A. A. Maciejewski, T. Braun, M. Schneider, S. Tideman, R. Chitta, R. B. Dilmaghani, R. Joshi, A. Kaul, A. Sharma, S. Sripada, P. Vangari, and S. S. Yellampalli, "Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment," *Journal of Parallel and Distributed Computing*, vol. 67, no. 2, pp. 154–169, Feb 2007.

[25] S. Ghanbari and M. R. Meybodi, "On-line mapping algorithms in highly heterogeneous computational grids: A learning automata approach," in *International Conference on Information and Knowledge Technology (IKT '05)*, May 2005.

[26] Q. Ding and G. Chen, "A benefit function mapping heuristic for a class of meta-tasks in grid environments," in *International Symposium on Cluster Computing and the Grid (CCGRID '01)*, May 2001, pp. 654–659.

[27] K. Kaya, B. Ucar, and C. Aykanat, "Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories," *Journal of Parallel and Distributed Computing*, vol. 67, no. 3, pp. 271–285, Mar. 2007.

[28] S. Shivle, H. J. Siegel, A. A. Maciejewski, P. Sugavanam, T. Banka, R. Castain, K. Chindam, S. Dussinger, P. Pichumani, P. Satyasekaran, W. Saylor, D. Sendek, J. Sousa, J. Sridharan, and J. Velazco, "Static allocation of resources to communicating subtasks in a heterogeneous ad hoc grid environment," *Journal of Parallel and Distributed Computing, Special Issue on Algorithms for Wireless and Ad-hoc Networks*, vol. 66, no. 4, pp. 600–611, Apr. 2006.

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

[30] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, S. Spring, A. Su, and D. Zagorodnov, "Adaptive computing on the grid using AppLeS," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 4, pp. 369–382, Apr. 2003.

[31] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for scheduling parameter sweep applications in grid environments," in *9th Heterogeneous Computing Workshop (HCW 2000)*, Mar. 2000, pp. 349–363.

[32] M. Snir and D. A. Bader, "A framework for measuring supercomputer productivity," *International Journal of High Performance Computing Applications*, vol. 18, no. 4, pp. 417–432, Nov. 2004.

[33] P. Li, B. Ravindran, H. Cho, and E. D. Jensen, "Scheduling distributable real-time threads in Tempus middleware," in *International Conference on Parallel and Distributed Systems (ICPADS '04)*, 2004, pp. 187–194.

[34] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 613–629, Sep. 2004.

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

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

[37] P. Bohrer, E. N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony, "The case for power management in web servers," in *Power Aware Computing*, ser. Series in Computer Science, R. Graybill and R. Melhem, Eds. Springer US, 2002.

[38] I. Rodero, J. Jaramillo, A. Quiroz, M. Parashar, F. Guim, and S. Poole, "Energy-efficient application-aware online provisioning for virtualized clouds and data centers," in *International Green Computing Conference*, Aug 2010, pp. 31–45.

[39] M. P. Mills. The Cloud Begins With Coal - Big Data, Big Networks, Big Infrastructure, and Big Power. Digital Power Group. [Online]. Available: http://www.tech-pundit.com/wp-content/uploads/2013/07/Cloud_Begins_With_Coal.pdf

[40] 2012 DatacenterDynamics Industry Census. [Online]. Available: http://www.datacenterdynamics.com/blogs/industry-census-2012-emerging-data-center-markets

[41] D. J. Brown and C. Reams, "Toward energy-efficient computing," *Communications of the ACM*, vol. 53, no. 3, pp. 50–58, Mar. 2010.

[42] A. M. Al-Qawasmeh, A. A. Maciejewski, R. G. Roberts, and H. J. Siegel, "Characterizing task-machine affinity in heterogeneous computing environments," in *20th Heterogeneity in Computing Workshop (HCW 2011), in the proceedings of the IPDPS 2011 Workshops & PhD Forum (IPDPSW)*, May 2011, pp. 33–43.

[43] "Introducing Titan," Jun 2014. [Online]. Available: https://www.olcf.ornl.gov/titan/

[44] H. Singh and A. Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," in *5th Heterogeneous Computing Workshop (HCW '96)*, Apr. 1996, pp. 86–97.

[45] R. Friese, B. Khemka, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, S. Powers, M. Hilton, J. Rambharos, G. Okonski, and S. W. Poole, "An analysis framework for investigating the trade-offs between system performance and energy consumption in a

heterogeneous computing environments," in *22nd Heterogeneity in Computing Workshop (HCW 2013), in the proceedings of the IPDPS 2013 Workshops & PhD Forum (IPDPSW)*, May 2013.

[46] Colorado State University ISTeC Cray High Performance Computing Systems. [Online]. Available: http://istec.colostate.edu/activities/cray

[47] Z. Jinquan, N. Lina, and J. Changjun, "A heuristic scheduling strategy for independent tasks on grid," in *International Conference on High-Performance Computing in Asia-Pacific Region*, Nov. 2005.

[48] Y. Tian, E. Ekici, and F. Ozguner, "Energy-constrained task mapping and scheduling in wireless sensor networks," in *IEEE Mobile Adhoc and Sensor Systems Conference*, Nov. 2005, p. 8.

[49] K. H. Kim, R. Buyya, and J. Kim, "Power aware scheduling of bag-of-tasks applications with deadline constraints on DVS-enabled clusters," in *IEEE/ACM International Symposium of Cluster Computing and the Grid (CCGrid 2007)*, 2007, pp. 541–548.

[50] R. Friese, T. Brinks, C. Oliver, A. A. Maciejewski, H. J. Siegel, and S. Pasricha, "A machine-by-machine analysis of a bi-objective resource allocation problems," in *The 2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2013)*, July 2013, pp. 3–9.

[51] B. Khemka, R. Friese, S. Pasricha, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, S. Powers, M. Hilton, R. Rambharos, M. Wright, and S. Poole, "Comparison of energy-constrained resource allocation heuristics under difference task management environments," in *accepted for publication, 2015 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2015)*, July 2015, p. 10.

[52] America's data centers consuming and wasting growing amounts of energy. [Online]. Available: http://www.nrdc.org/energy/data-center-efficiency-assessment.asp

[53] Advanced configuration and power interface specification. [Online]. Available: http://www.acpi.info/spec.htm

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

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

[56] H. Yu, B. Veeravalli, and Y. Ha, "Dynamic scheduling of imprecise-computation tasks in maximizing QoS under energy constraints for embedded systems," in *Asia and South Pacific Design Automation Conference (ASPDAC 2008)*, Mar. 2008, pp. 452–455.

[57] Environmental Protection Agency, "Report to congress on server and data center energy efficency," http://www.energystar.gov/ia/partners/prod_development/downloads/EPA_Datacenter_Report_Congress_Final1.pdf, Aug. 2007.

[58] J. Koomey, "Growth in data center electricity use 2005 to 2010," *Analytics Press*, Aug. 2011.

[59] R. Friese, T. Brinks, C. Oliver, H. J. Siegel, and A. A. Maciejewski, "Analyzing the trade-offs between minimizing makespan and minimizing energy consumption in a heterogeneous resource allocation problem," in *The 2nd International Conference on Advanced Communications and Computation (INFOCOMP 2012)*, Oct. 2012, p. 9.

[60] L. D. Briceño, B. Khemka, H. J. Siegel, A. A. Maciejewski, C. Groer, G. Koenig, G. Okonski, and S. Poole, "Time utility functions for modeling and evaluating resource allocations in a heterogeneous computing systems," in *20th Heterogeneity in Computing*

Workshop (HCW 2011), in the proceedings of the IPDPS 2011 Workshops & PhD Forum (IPDPSW), May 2011, pp. 7–19.

[61] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[62] J. J. Dongarra, E. Jeannot, E. Saule, and Z. Shi, "Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems," in *The 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '07)*, 2007, pp. 280–288.

[63] E. Jeannot, E. Saule, and D. Trystram, "Bi-objective approximation scheme for makespan and reliability optimization on uniform parallel machines," in *The 14th International Euro-Par Conference on Parallel Processing (Euro-Par '08)*, vol. 5168, 2008, pp. 877–886.

[64] B. Abbasi, S. Shadrokh, and J. Arkat, "Bi-objective resource-constrained project scheduling with robustness and makespan criteria," *Applied Mathematics and Computation*, vol. 180, no. 1, pp. 146–152, 2006.

[65] J. Pasia, R. Hartl, and K. Doerner, "Solving a bi-objective flowshop scheduling problem by Pareto-ant colony optimization," in *Ant Colony Optimization and Swarm Intelligence*, vol. 4150, 2006, pp. 294–305.

[66] Y. He, F. Liu, H.-j. Cao, and C.-b. Li, "A bi-objective model for job-shop scheduling problem to minimize both energy consumption and makespan," *Journal of Central South University of Technology*, vol. 12, pp. 167–171, Oct. 2005.

[67] 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 *9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA '11)*, Dec. 2011, pp. 22–31.

[68] J. Teich, "Hardware/software codesign: The past, the present, and predicting the future," *Proceedings of the IEEE*, vol. 100, no. 13, pp. 1411–1430, 2012.

[69] (accessed: 07/24/2012) Intel core i7 3770k power consumption, thermal. [Online]. Available: http://openbenchmarking.org/result/1204229-SU-CPUMONITO81#system_table

[70] A. M. Al-Qawasmeh, A. A. Maciejewski, H. Wang, J. Smith, H. J. Siegel, and J. Potter, "Statistical measures for quantifying task and machine heterogeneities," *The Journal of Supercomputing, Special Issue on Advances in Parallel and Distributed Computing*, vol. 57, no. 1, pp. 34–50, July 2011.

[71] M. G. Kendall, *The Advanced Theory of Statistics*. Charles Griffin and Company Limited, 1945, vol. 1.

[72] V. Pareto, *Cours d'economie politique*. Lausanne: F. Rouge, 1896.

[73] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms: A tutorial," *Journal of Reliability Engineering ans System Safety*, vol. 91, pp. 992–1007, Sept 2006.

[74] Y. He, F. Liu, H.-j. Cao, and C.-b. Li, "A bi-objective model for job-shop scheduling problem to minimize both energy consumption and makespan," *Journal of Central South University of Technology*, vol. 12, pp. 167–171, Oct 2005.

[75] P. Chitra, R. Rajaram, and P. Venkatesh, "Application and comparison of hybrid evolutionary multiobjective optimization algorithms for solving task scheduling problem on

heterogeneous systems," *Applied Soft Computing*, vol. 11, no. 2, pp. 2725–2734, Mar 2011.

[76] P. Lindberg, J. Leingang, D. Lysaker, S. U. Khan, and J. Li, "Comparison and analysis of eight scheduling heuristics for the optimization of energy consumption and makespan in large-scale distributed systems," *Supercomputing*, vol. 59, no. 1, pp. 323–360, Jan 2012.

[77] G. Bernat, A. Colin, and S. Petters, "WCET analysis of probabilistic hard real-time systems," in *23rd IEEE Real-Time Systems Symposium*, Dec. 2002, pp. 279–288.

[78] L. David and I. Puaut, "Static determination of probabilistic execution times," in *16th Euromicro Conference on Real-Time Systems*, 2004, pp. 223–230.

[79] 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," *Journal of Parallel and Distributed Computing*, vol. 44, no. 1, pp. 35–52, 1997.

[80] D. Dauwe, R. Friese, S. Pasricha, A. A. Maciejewski, G. A. Koenig, and H. J. Siegel, "Modeling the effects on power and performance from memory interference of co-located applications in multicore systems," in *The 2014 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2014)*, Las Vegas, NV, July 2014, pp. 3–9.

[81] (accessed Mar. 2014) PARSEC benchmark suite. [Online]. Available: http://parsec.cs.princeton.edu/

[82] M. Oxley, S. Pasricha, H. J. Siegel, and A. A. Maciejewski, "Energy and deadline constrained robust stochastic static resource alloation," in *1st Workshop on Power and*

Energy Aspects of Computation (PEAC 2013), in the proceedings of the 10th International Conference on Parallel Processing and Applied Mathematics (PPAM 2013), Sep 2013, p. 10.

[83] S. Kim, C. Choi, H. Eom, and H. Y. Yeom, "Energy-centric DVFS controlling method for multi-core platforms," in *5th International Workshop on Multi-Core Computing Systems (MuCoCoS'12), as part of Super Computing 2012*, Nov 2012.

[84] E. L. Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *The 2010 International Conference on Power Aware Computing and Systems (HotPower '10)*, Oct 2010, p. 5.

[85] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *38th Annual International Symposium on Computer Architecture (ISCA'11)*, June 2011, pp. 283–294.

[86] A. Jaleel, "Memory characterization of workloads using intrumentation-driven simulation," Tech. Rep., 2007. [Online]. Available: http://www.jaleels.org/ajaleel/workload/SPECanalysis.pdf

[87] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *2nd Symposium on Cloud Computing (SOCC'11)*, 2011, pp. 1–14.

[88] "Intel 64 and ia-32 architectures software developer's manual volume 3b: System programming guide, part 2," Tech. Rep., Feb 2014. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf

[89] (accessed Mar. 2014) Performance application programming interface. [Online]. Available: http://icl.cs.utk.edu/papi/

[90] (accessed Mar. 2014) PAPI events by architectures. [Online]. Available: http://icl.cs.utk.edu/projects/papi/presets.html

[91] (accessed Mar. 2014) HPCToolkit. [Online]. Available: http://hpctoolkit.org/

[92] (accessed Mar. 2014) Watts Up? plug load meters. [Online]. Available: https://www.wattsupmeters.com/secure/products.php?pn=0

[93] Intel. (2012) Intel core i7-3770 processor. [Online]. Available: http://ark.intel.com/products/65719/

[94] I. Ahmad and A. Ghafoor, "Semi-distributed load balancing for massively parallel multicomputer systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp. 987–1004, Oct 1991.

[95] D. Young, S. Pasricha, A. A. Maciejewski, H. J. Siegel, and J. T. Smith, "Heterogeneous energy and makespan-constrained dag scheduling," in *International Workshop on Energy Efficient High Performance Parallel and Distributed Computing (EEHPDC-2013)*, Jun. 2013, pp. 3–11.

APPENDIX A

Permuting Initial Virtual-Queue Tasks

For the batch-mode heuristics, we experiment with an additional technique to modify the ordering of the tasks that are at the head of the virtual queue of the machines. The motivation for doing this is to take advantage of the following situation: consider that there are two tasks, with one having a utility function that starts at a higher utility value but decays very slowly (marked as "1" in Fig. A.1), whereas the other task's utility function starts at a relatively lower utility, but has a quick decay occurring very soon (marked as "2" in Fig. A.1). The utility maximizing heuristics may schedule task 1 ahead of task 2 when assigning them to a machine. In this scenario, higher overall utility may be earned by switching the execution order because the loss in utility from task 1's delayed execution might be less than the gain in utility from the earlier completion of task 2.

To capture this benefit, once the batch-mode heuristic has made its resource allocation decisions, we try all permutations of the two, three, or five initial tasks within the virtual queue of each machine. For each machine, the ordering that earns the highest utility from the three tasks is chosen. We use a dropping threshold of 0.5 to perform our experiments with the permuting operation because it explicitly does not drop tasks of certain priority levels. There was no significant difference in the performance of the batch-mode heuristics with the permuting operation.

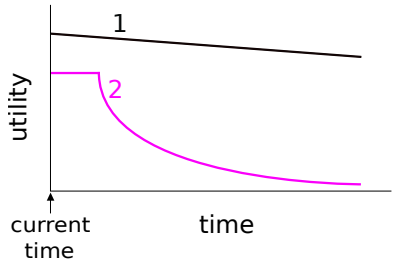


FIGURE A.1. Though task 2 has lower utility than task 1, there might be benefit in scheduling it before task 1.

APPENDIX B

Calculating Duration of the First Interval

To make the starting utility value of each task realizable, this utility value persists for some time before it starts to decay. If this is not done, even if the task starts execution as soon as it arrives on the machine that can complete it the soonest, it will not be possible to obtain the task's maximum utility value. For our simulation studies, the length of time for which the starting utility value of the task persists is dependent on the urgency level of the task and its average execution time across the machine types. We compute for each task the average execution time of that task across the machine types (ignoring machine types that cannot execute it). We scale this average value by a factor dependent on the urgency level of the task. If the task is an *extreme* urgency task, then we scale the average execution time of this task by 80% to obtain the length of its first interval. For *high*, *medium*, and *low* urgency levels, we scale the average value by 90%, 100% (no scaling), and 110%, respectively.

APPENDIX C

Values of the Utility Classes

As mentioned in Sec. 2.2.1.2, a utility class has three parameters associated with each interval (except the first). The values of the three parameters used to create the four utility classes that we model in this study are given Table C.1. The utility of all utility classes drops to zero 10,000 minutes after their first interval has completed.

Table C.1. Values of the three parameters for the different intervals (except the first) of the four utility classes that we model in this study. τ is the arrival time of the task and F is the duration of the first interval.

Utility Class	**Parameter**	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$
A	$t(k, A)$	$\tau + F$	$\tau + F + 5$	$\tau + F + 10$	$\tau + F + 20$	$\tau + F + 30$	$\tau + F + 10000$
	$\psi(k, A)$	100%	60%	30%	20%	10%	0%
	$\delta(k, A)$	1.1	1.15	1.2	1.1	1.2	10
B	$t(k, B)$	$\tau + F$	$\tau + F + 7$	$\tau + F + 15$	$\tau + F + 22.5$	$\tau + F + 30$	$\tau + F + 10000$
	$\psi(k, B)$	100%	50%	25%	12%	5%	0%
	$\delta(k, B)$	0.9	0.9	0.9	0.9	0.9	10
C	$t(k, C)$	$\tau + F$	$\tau + F + 10$	$\tau + F + 20$	$\tau + F + 30$	$\tau + F + 40$	$\tau + F + 10000$
	$\psi(k, C)$	100%	75%	50%	25%	12%	0%
	$\delta(k, C)$	0.9	0.85	0.85	0.8	0.8	10
D	$t(k, D)$	$\tau + F$	$\tau + F + 12.5$	$\tau + F + 25$	$\tau + F + 37$	$\tau + F + 50$	$\tau + F + 10000$
	$\psi(k, D)$	100%	80%	66%	33%	11%	0%
	$\delta(k, D)$	1.2	1.1	1.1	0.9	0.9	10

APPENDIX D

Joint Probability Distribution of Priority and Urgency Levels

To generate the priority and urgency levels for the tasks, we use a joint probability distribution represented by the matrix shown in Table D.1. This table is representative of DOE/DoD environments. The matrix models an environment where the probability of a task having *critical* priority and *low* urgency is zero. Similarly, *extreme* and *high* urgency tasks are unlikely to have *low* priority. Most of the tasks have *medium* and *low* priorities with *medium* and *low* urgencies. A few important tasks have *critical* and *high* priorities with *extreme* and *high* urgencies. The results in Sec. 2.7 show that the timely execution of the *critical* priority tasks (approximately 4% of the tasks) significantly contributes to the total utility earned by the system. For each task, we sample from this joint probability distribution to obtain the task's priority and urgency levels.

Table D.1. The joint probability distribution of tasks having certain priority and urgency levels

priority levels	**urgency levels**			
	extreme	***high***	***medium***	***low***
critical	2%	2%	0.05%	0%
high	3.45%	5%	1.5%	3%
medium	0%	10%	10%	10%
low	0%	0%	20%	33%

APPENDIX E

Simulation Parameters for Generating Estimated Time to Compute (ETC) Matrices

To generate the entries of the ETC matrix, we adopt the Coefficient of Variation (COV) method [35] to our environment. The mean value of execution time on the general-purpose and the special-purpose machine types is set to ten minutes and one minute, respectively. The coefficient of variation along the task types is set to 0.1. The coefficient of variation along the special-purpose machine types is also set to 0.1, whereas the coefficient of variation along the general-purpose machine types is set to 0.25. This models heterogeneity in the ETC matrix [35]. To represent the fact that some task types are incapable of executing on some of the special-purpose machine types, we set the corresponding entries of the matrix to infinity. Table E.1 shows a sample ETC matrix with only four machine types and only four task types. Among the four machine types, machine types A and B are modeled as special-purpose machine types. Each of them has one task type that is special on them.

Across the different simulation trials, the actual number of machines for each machine type is constant, but the properties of the machine types varies. The partitioning of the 100 machines into the 13 machine types is as follows: 2, 2, 3, 3, 5, 5, 5, 10, 10, 10, 10, 15, and 20. The first four machine types in this list are the special-purpose machine types. So, in all we have 10 special-purpose machines. This distribution of the number of machines across the machine types is chosen based on the expectations for future environments of DOE and DoD interest.

TABLE E.1. A sample ETC matrix with only four machine types and only four task types showing the execution times in minutes. Machine types A and B are special-purpose machine types (task types 1 and 2, respectively, are special on them). All other task types are incompatible on the special-purpose machine types. In the table, "spl" is used to denote a special-purpose task/machine type and "gen" is used to denote a general-purpose task/machine type.

	machine type A (spl)	**machine type B (spl)**	**machine type C (gen)**	**machine type D (gen)**
task type 1 (spl)	1.1	∞	13	9
task type 2 (spl)	∞	0.9	8	11
task type 3 (gen)	∞	∞	10	12
task type 4 (gen)	∞	∞	12	9

APPENDIX F

Generation of Task Arrivals for Simulations

The arrival patterns used are based on the expectations for future environments of DOE and DoD interest. For the general-purpose task types, we use a sinusoidal pattern for the arrival rate. We set the frequency of the sinusoidal curve by specifying the number of complete sinusoidal cycles to occur during the 24 hour period. For each general-purpose task type, we randomly select an integer from 1 to 24 with uniform probability to obtain the number of sinusoidal cycles. We do not use fractions because the integers ensure that at the start and end of the 24 hour period (i.e., end of the 2^{nd} and end of the 26^{th} hour), the arrival rates are equal. This is important because the arrival pattern models a day and, the end of the 2^{nd} and the 26^{th} hour correspond to the same time of the day. The phase-shift of the sinusoidal curve is randomly sampled from the range 0 to 2π using uniform probability. The amplitude is calculated by multiplying an amplitude factor and the mean arrival rate. The amplitude factor of the curve is sampled randomly from the range 0.25 to 0.9. Using this technique, each general-purpose task type has its own arrival rate pattern. Fig. F.1 shows example sinusoidal arrival rate patterns (with dashed lines showing their mean arrival rates) for five general-purpose task types.

For the special-purpose task types, we use a "bursty" arrival rate pattern. The pattern consists of two types of alternating intervals for the arrival rate: baseline interval and the burst interval. The baseline intervals have a lower arrival rate and a longer duration than the burst intervals. For each baseline interval, the arrival rate is obtained by multiplying the mean arrival rate (computed as mentioned before) with a number sampled uniformly at random from the range [0.5, 0.75]. In contrast, for the burst interval the range from

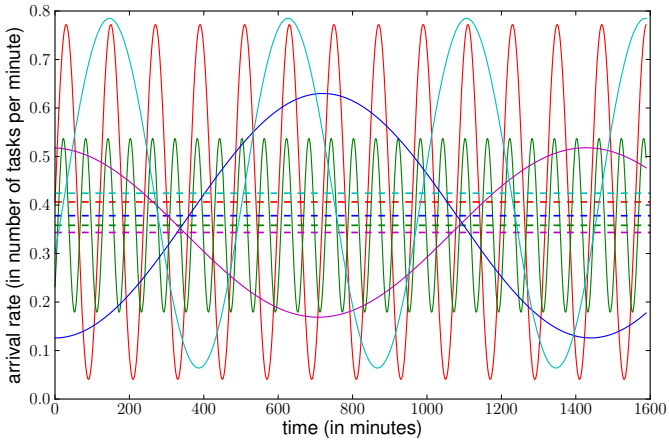


FIGURE F.1. Example sinusoidal curves that model the arrival rate for the general-purpose task types. Curves for five general-purpose task types are shown with dashed lines representing their mean arrival rates.

which the random number is sampled is [1.25, 1.5]. The duration of each baseline interval is obtained by sampling uniformly at random from the range [3, 5] hours, whereas for each burst interval the range is [30, 90] minutes. To ensure that the arrival rate of the 2^{nd} and 26^{th} hour remain the same, we make sure that the duration of the interval that was present just before the end of the 2^{nd} hour is repeated just before the end of the 26^{th} hour. Fig. F.2 shows example arrival rate patterns for five special-purpose task types with their mean arrival rate shown using dashed lines.

Once we have an arrival rate pattern for every task type, we step along the curve to generate the arrival times of the different tasks that will belong to this task type. We start with the arrival rate at the beginning of each curve. We sample an exponential distribution with the rate to get a time duration. We step along the curve based on the sampled time duration value and generate the arrival time of the next task (of this type). We keep repeating the process until the end of the 26^{th} hour. This generates not only the arrival times but also generates the number of tasks that belong to that task type. In regions where the arrival rate is higher, the sampled time from the exponential distribution is lower, and therefore the

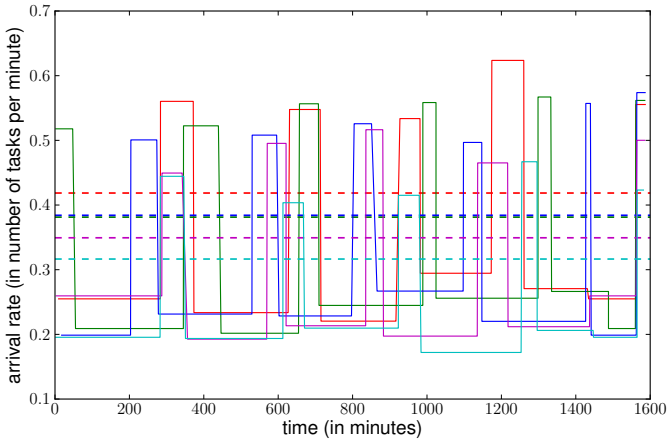


FIGURE F.2. Step-shaped curves that model the baseline and burst periods of arrival rates for the special-purpose task types. Example curves for five special-purpose task types are shown with dashed lines representing their mean arrival rates.

arrival time of the next task is closer to the current task's arrival time. If the arrival rate is very low for some part of the arrival rate curve, then the time value sampled from the exponential distribution might be too long. This may prevent any further sampling for this task type. To avoid such cases, each time we sample from the exponential distribution, if the sampled next arrival time of a task is greater than a pre-set upper limit, then we set the next arrival time to that upper limit value. For our simulations we set the upper limit to $1/50^{th}$ of the 24 hour duration, i.e., $24 \times 60/50 = 28.8$ minutes. Fig. F.3 shows the number of tasks that arrive in a minute (including both general-purpose and special-purpose) from a single simulation trial.

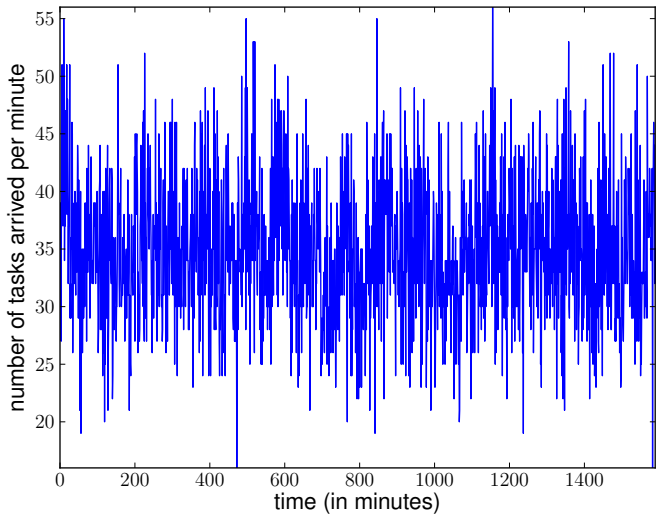


FIGURE F.3. An example trace of the number of tasks (both general-purpose and special-purpose) that arrive per minute as a function of time. We generate the arrival of tasks for a duration of 26 hours.

APPENDIX G

Results from 33,000 Tasks per Day Oversubscription Level

Fig. G.1 shows the results with the dropping operation for all the heuristics when we had 33,000 tasks arrive in the day. The two notable differences between the percentage of maximum utility earned by the heuristics in the 33,000 tasks per day case as opposed to the 50,000 tasks per day case are that all the heuristics are able to earn a higher percentage of maximum utility (because the environment is not as oversubscribed), and the batch-mode heuristics do not have as much of an increase in performance with the dropping operation (because even with the no dropping case, on average only 32% of mapping events were delayed due to excessive heuristic execution times).

Table G.1 gives the average execution time of the mapping events for the heuristics with a dropping threshold of 0.5. These times include the execution time for the heuristic and the dropping operation.

Table G.1. Average execution time of the mapping events for all the heuristics with a dropping threshold of 0.5 for the two levels of oversubscription.

heuristic	**mapping event execution time (in milliseconds)**	
	33,000 tasks per day	**50,000 tasks per day**
Random	0.14	0.15
Round-Robin	0.14	0.15
Max Util	0.20	0.22
Max UPT	0.30	0.33
MET-Random	0.20	0.22
MET-Max Util	0.20	0.22
Min-Min Comp	9.76	46.66
Sufferage	57.32	280.09
Max-Max Util	51.64	316.47
Max-Max UPT	64.2	319.61
MET-Max Util-Max UPT	23.68	66.22

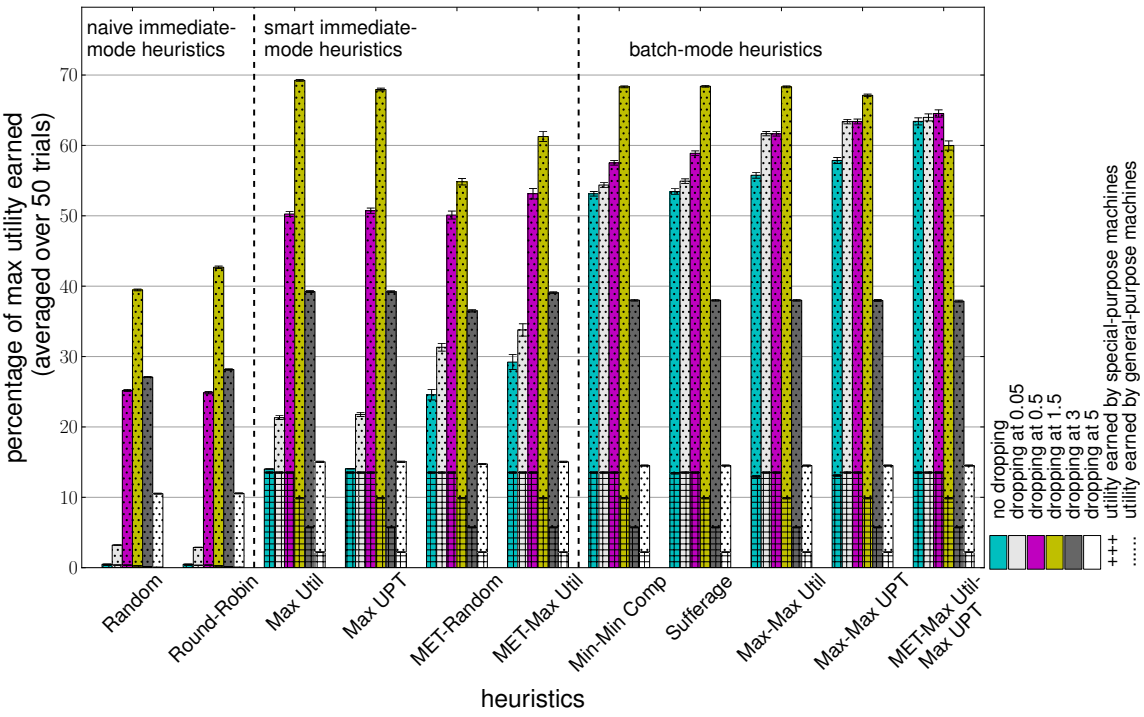


FIGURE G.1. Percentage of maximum utility earned by all the heuristics for the different dropping thresholds with an oversubscription level of 33,000 tasks arriving during the day. The average maximum utility bound for this oversubscription level is 65,051.

APPENDIX H

Discussion of Additional Results

We performed experiments with the maximum utility values for the priority levels set at 1000, 100, 10, and 1 instead of 8, 4, 2, and 1, respectively. The dropping thresholds that we used in that case were: 500, 50, 5, 0.5, 0.05, and 0.005. We observed that the utility being earned in those cases was controlled to a large extent by the timely execution of the *critical* priority tasks. A significant amount of utility could be earned even if all tasks except the *critical* priority tasks were dropped. This is because with the priorities set at 1000, 100, 10, and 1, it takes ten *high* priority tasks to equal the benefit of one *critical* priority task, as opposed to two *high* priority tasks in our current model with priorities set at 8, 4, 2, and 1. These latter priority values better match the intended environment.