Energy-Efficient Resource Management Techniques for Cloud Data Centers

Abstract approved: _______________________________________________________________________

Bechir Hamdaoui

Energy consumption has become a great deal for cloud service providers due to financial as well as environmental concerns. Studies show that cloud servers operate, most of the time, at only between 10% and 50% of their maximal utilizations. These same studies also show that servers that are kept ON but are idle or lightly utilized consume significant amounts of energy. In this thesis, we develop new resource management techniques that enable automated, energy-efficient allocation of cloud data center resources, thereby reducing energy consumption and hence cutting down on the electricity bills of service providers. Specifically, our developed techniques consist of the following four complimentary frameworks:

1. **Workload Prediction Framework.** It predicts the number of Virtual Machine (VM) requests along with their amounts of CPU and memory resources using k-Means clustering and adaptive Wiener filter prediction techniques. This proposed prediction framework provides accurate estimations of the number of needed servers, thus reducing energy consumption by putting to sleep unneeded servers.
2. Resource Scheduling Framework. It reduces the time during which servers are kept ON by placing VMs with similar completion times on the same server while allocating more resources to the VMs that need more time to accomplish their tasks. The framework also allocates more resources for the delay-sensitive tasks whose charging cost is dependent on how fast they accomplish so that they finish earlier which generates higher revenues. This is all done by solving a convex optimization problem that guarantees that all the scheduled tasks meet their hard deadlines.

3. Resource Overcommitment Framework. It consolidates data center workloads on as few ON servers as possible by assigning VMs to a server in excess of its real capacity, anticipating that each assigned VM will only utilize a part of its requested resources. The framework first determines the amount of server resources that can be overcommitted by predicting the future resource demands of admitted VMs. It then handles server overloads by predicting them before occurring and migrating VMs from overloaded servers to other under-utilized or idle servers whenever an overload occurs or is predicted to occur.

4. Peak Shaving Control Framework. It spreads the data center’s power demands more evenly over the entire billing cycle by making smart workload shifting and energy storage decisions which leads into great monetary reductions in the grid’s peak demand penalties. The framework accounts for real energy storage losses and constraints, and considers a heterogeneous cloud workload that is made up of multiple classes, with each class having different delay tolerance and price.

Several experiments based on real traces from a Google cluster show that our proposed frameworks achieve significant utilization gains, energy reductions, and monetary savings when compared to state-of-the-art cloud resource management techniques.

APPROVED:

__________________________
Major Professor, representing Electrical and Computer Engineering

__________________________
Director of the School of Electrical Engineering and Computer Science

__________________________
Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

__________________________
Mehiar Dabbagh, Author
To the many that influenced my life, namely my family, my friends, and my mentors
I am very grateful for all the knowledge and experience I gained, the opportunities I obtained and the friendships I made within my PhD journey. I would not have completed this journey without the help and support of many people. First, my deepest gratitude goes to my adviser Professor Bechir Hamdaoui for his guidance, his mentorship, and his unlimited support.

I would like to express my gratitude to the committee members, Professors Attila Yavuz, Bella Bose, Eduardo Cotilla-Sanchez, and to the GCR representatives Professors Maggie Niess and James Strother for their helpful discussions and constructive feedback. I also thank Professor Mohsen Guizani for his valuable assistance with this work.

I am very thankful for the internship opportunities that Gert Drapers and Professor Ammar Rayes offered me within my PhD, which helped me obtain new skills and experiences. I am also grateful to the Institute of International Education (IIE) for partially funding me in my last term of my PhD. The courses I took at Oregon State University and at the American University of Beirut have helped me a lot with my thesis work so I am very thankful for everyone who taught me in these two institutes and especially my previous adviser Professor Hazem Hajj.

I am heartily thankful to my parents and to my sister for their unconditional love and their continuous encouragement. Without their support I would not have made it here. Finally I thank my friends Majd, Cesar, Doaa, and my roommates and office colleagues for making my graduate years exciting and pleasurable.
# TABLE OF CONTENTS

1 Introduction

1.1 Why Worry About Energy? ................................................. 1
1.2 The Cloud Paradigm ..................................................... 2
1.3 New Challenges, New Opportunities ................................. 2
1.4 Dissertation Contributions and Organization .................. 4

2 Energy-Efficient Workload Prediction and Resource Provisioning Framework 6

2.1 INTRODUCTION .......................................................... 6
2.2 RELATED WORK .......................................................... 9
  2.2.1 Server Level Techniques ......................................... 9
  2.2.2 Data Center Level Techniques ................................. 11
2.3 THE PROPOSED FRAMEWORK ........................................... 14
  2.3.1 Data Clustering .................................................... 14
  2.3.2 Workload Prediction .............................................. 17
  2.3.3 Power Management ............................................... 17
2.4 DATA CLUSTERING ........................................................ 19
  2.4.1 Google Traces ..................................................... 19
  2.4.2 Workload Clustering ............................................. 22
2.5 WORKLOAD PREDICTION ................................................. 25
  2.5.1 Length of the Prediction Window .............................. 25
  2.5.2 Weights of the Stochastic Predictor ......................... 27
  2.5.3 Length of the Observation Window ............................ 30
2.6 PREDICTION ENHANCEMENTS ......................................... 32
  2.6.1 Adaptive Predictor ............................................... 32
  2.6.2 Prediction Accuracy ............................................. 34
  2.6.3 Safety Margin .................................................... 36
2.7 POWER MANAGEMENT .................................................... 39
2.8 FRAMEWORK EVALUATION ............................................. 42
  2.8.1 Resource Utilization ............................................. 44
  2.8.2 Energy Savings .................................................. 47
  2.8.3 Workload Underestimation Evaluation ....................... 49
  2.8.4 Execution Time Overhead Evaluation ....................... 49
2.9 CONCLUSION AND FUTURE WORK ..................................... 50
# TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Energy Efficient Resource Allocation and Scheduling Framework</td>
<td>52</td>
</tr>
<tr>
<td>3.1 Related Work</td>
<td>52</td>
</tr>
<tr>
<td>3.2 The Proposed Framework</td>
<td>56</td>
</tr>
<tr>
<td>3.2.1 VM Placement Module</td>
<td>56</td>
</tr>
<tr>
<td>3.2.2 Resource Management Module</td>
<td>58</td>
</tr>
<tr>
<td>3.3 Charging Models</td>
<td>61</td>
</tr>
<tr>
<td>3.4 PM Resource Allocation</td>
<td>63</td>
</tr>
<tr>
<td>3.4.1 Formulated Optimization Problem</td>
<td>64</td>
</tr>
<tr>
<td>3.4.2 Equivalent Optimization Problem</td>
<td>66</td>
</tr>
<tr>
<td>3.5 Framework Evaluation</td>
<td>69</td>
</tr>
<tr>
<td>3.5.1 Best-Effort Service</td>
<td>71</td>
</tr>
<tr>
<td>3.5.2 Best-Effort and Delay-Sensitive Services</td>
<td>77</td>
</tr>
<tr>
<td>3.6 Conclusions</td>
<td>80</td>
</tr>
<tr>
<td>3.7 Future Work Directions</td>
<td>80</td>
</tr>
<tr>
<td>4 Energy-Efficient Resource Overcommitment Framework</td>
<td>82</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>82</td>
</tr>
<tr>
<td>4.2 Related Work</td>
<td>85</td>
</tr>
<tr>
<td>4.2.1 Overload Avoidance Techniques</td>
<td>86</td>
</tr>
<tr>
<td>4.2.2 VM Migration Techniques</td>
<td>87</td>
</tr>
<tr>
<td>4.3 Proposed Framework</td>
<td>88</td>
</tr>
<tr>
<td>4.3.1 Resource Predictor</td>
<td>90</td>
</tr>
<tr>
<td>4.3.2 Overload Predictor</td>
<td>90</td>
</tr>
<tr>
<td>4.3.3 Cluster Manager</td>
<td>92</td>
</tr>
<tr>
<td>4.4 VM Resource Predictor</td>
<td>94</td>
</tr>
<tr>
<td>4.5 Energy-Aware VM Migration</td>
<td>98</td>
</tr>
<tr>
<td>4.6 Proposed Heuristic</td>
<td>101</td>
</tr>
<tr>
<td>4.7 Framework Evaluation</td>
<td>103</td>
</tr>
<tr>
<td>4.8 Conclusion and Future Work</td>
<td>116</td>
</tr>
<tr>
<td>5 Workload Shifting and Energy Storage Peak Shaving Framework</td>
<td>117</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>117</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>5.2</td>
<td>The Case for Google DC</td>
</tr>
<tr>
<td>5.3</td>
<td>Notations</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Energy Storage Notations</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Battery Specs</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Workload Shifting Notations</td>
</tr>
<tr>
<td>5.4</td>
<td>Full-Horizon Optimal Control</td>
</tr>
<tr>
<td>5.5</td>
<td>Limited-Horizon Control</td>
</tr>
<tr>
<td>5.6</td>
<td>Evaluation</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Peak Shaving Through Energy Storage</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Peak Shaving Through Workload Shifting</td>
</tr>
<tr>
<td>5.6.3</td>
<td>Peak Shaving Through Energy Storage &amp; Workload Shifting</td>
</tr>
<tr>
<td>5.7</td>
<td>Related Work</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Energy Charge Minimization Techniques</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Energy Storage Peak Shaving Techniques</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Workload Modulation Peak Shaving Techniques</td>
</tr>
<tr>
<td>5.8</td>
<td>Conclusion and Future Work</td>
</tr>
<tr>
<td>6</td>
<td>Dissertation Conclusions</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Flow chart of the proposed framework</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>The elbow criteria for selecting $k$.</td>
<td>23</td>
</tr>
<tr>
<td>2.3</td>
<td>The resulting four clusters/categories for Google traces.</td>
<td>24</td>
</tr>
<tr>
<td>2.4</td>
<td>The general structure of Wiener filter.</td>
<td>27</td>
</tr>
<tr>
<td>2.5</td>
<td>RMSE for 3rd category predictor</td>
<td>30</td>
</tr>
<tr>
<td>2.6</td>
<td>Static versus adaptive Wiener filter prediction for category 3.</td>
<td>34</td>
</tr>
<tr>
<td>2.7</td>
<td>RMSE comparison of different predictive approaches.</td>
<td>36</td>
</tr>
<tr>
<td>2.8</td>
<td>Adaptive prediction with and without safety margin for category 2.</td>
<td>38</td>
</tr>
<tr>
<td>2.9</td>
<td>Adaptive prediction with and without safety margin for category 3.</td>
<td>38</td>
</tr>
<tr>
<td>2.10</td>
<td>Total number of unpredicted waken PMs during the entire testing period.</td>
<td>40</td>
</tr>
<tr>
<td>2.11</td>
<td>A comparison of the average CPU utilization.</td>
<td>45</td>
</tr>
<tr>
<td>2.12</td>
<td>A comparison of the average memory utilization.</td>
<td>45</td>
</tr>
<tr>
<td>2.13</td>
<td>A comparison of the consumed energy.</td>
<td>46</td>
</tr>
<tr>
<td>2.14</td>
<td>Total consumed energy during the entire testing period.</td>
<td>46</td>
</tr>
<tr>
<td>3.1</td>
<td>Two feasible allocations for the illustrative example.</td>
<td>54</td>
</tr>
<tr>
<td>3.2</td>
<td>Proposed Resource Allocation Framework.</td>
<td>56</td>
</tr>
<tr>
<td>3.3</td>
<td>Proposed charging models for elastic tasks.</td>
<td>63</td>
</tr>
<tr>
<td>3.4</td>
<td>Number of ON PMs over time for the different resource allocation schemes.</td>
<td>73</td>
</tr>
<tr>
<td>3.5</td>
<td>Average Utilization over time for the different resource allocation schemes.</td>
<td>73</td>
</tr>
<tr>
<td>3.6</td>
<td>Total Energy Consumption for the different schemes (normalized w.r.t. the energy consumption of BF Max).</td>
<td>74</td>
</tr>
<tr>
<td>3.7</td>
<td>Total cluster energy under different percentage of elastic tasks when $\lambda = 0.5$.</td>
<td>76</td>
</tr>
<tr>
<td>3.8</td>
<td>Total cluster energy under different elasticity factors when $\rho = 100%$.</td>
<td>76</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.9</td>
<td>The collected revenues for the different schemes (normalized w.r.t. Max BF revenues)</td>
<td>78</td>
</tr>
<tr>
<td>3.10</td>
<td>The energy expenses for the different schemes (normalized w.r.t. Max BF expenses)</td>
<td>78</td>
</tr>
<tr>
<td>3.11</td>
<td>The profits for the different schemes (normalized w.r.t. our framework’s profits)</td>
<td>78</td>
</tr>
<tr>
<td>4.1</td>
<td>One-day snapshot of Google cluster traces: percentage of utilized resources among all reserved resources.</td>
<td>84</td>
</tr>
<tr>
<td>4.2</td>
<td>Flow chart of the proposed framework</td>
<td>89</td>
</tr>
<tr>
<td>4.3</td>
<td>Number of overloads that were predicted and that were unpredicted by our framework over time.</td>
<td>105</td>
</tr>
<tr>
<td>4.4</td>
<td>Total number of unpredicted overloads during entire testing period using different overload avoidance techniques.</td>
<td>105</td>
</tr>
<tr>
<td>4.5</td>
<td>Total migration energy overhead under each of the three heuristics.</td>
<td>106</td>
</tr>
<tr>
<td>4.6</td>
<td>VM moving energy overhead.</td>
<td>106</td>
</tr>
<tr>
<td>4.7</td>
<td>The number of waken PMs to host the moved VMs.</td>
<td>106</td>
</tr>
<tr>
<td>4.8</td>
<td>Total migration overhead aggregated over the entire testing period (normalized w.r.t. Enhanced Largest First overheads).</td>
<td>107</td>
</tr>
<tr>
<td>4.9</td>
<td>Total migration overhead for the proposed migration heuristic under different values of $N_{on}$ and $N_{sleep}$ (normalized w.r.t. Lower Bound).</td>
<td>107</td>
</tr>
<tr>
<td>4.10</td>
<td>Number of PMs needed to host the workload.</td>
<td>107</td>
</tr>
<tr>
<td>4.11</td>
<td>Amount of energy that our allocation framework saves every hour when compared to allocation without overcommitment.</td>
<td>108</td>
</tr>
<tr>
<td>4.12</td>
<td>Google cluster’s total consumed energy during the 24-hour trace period for the different schemes.</td>
<td>108</td>
</tr>
<tr>
<td>5.1</td>
<td>The power drawn from the Grid by Google DC.</td>
<td>122</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>5.2</td>
<td>Breakdown of Google DC total electricity bill (based on the power consumption in Fig. 5.1)</td>
<td>123</td>
</tr>
<tr>
<td>5.3</td>
<td>Illustration of the power flow</td>
<td>126</td>
</tr>
<tr>
<td>5.4</td>
<td>Temporal illustration of the prediction window</td>
<td>132</td>
</tr>
<tr>
<td>5.5</td>
<td>Full-horizon control monetary savings for the different types of batteries based on Google traces</td>
<td>139</td>
</tr>
<tr>
<td>5.6</td>
<td>Google DC total electricity bill for different battery technologies that are operating by our proposed full-horizon controller under different energy storage capacities</td>
<td>139</td>
</tr>
<tr>
<td>5.7</td>
<td>Comparative evaluation of the different control techniques for each battery type based on Google traces</td>
<td>142</td>
</tr>
<tr>
<td>5.8</td>
<td>The electricity bill of Google DC as the length of the Prediction Window increases for the Limited-Horizon (Oracle) controller when operating different types of batteries</td>
<td>143</td>
</tr>
<tr>
<td>5.9</td>
<td>The time that the Limited-Horizon controller took to make decisions for different lengths of Prediction Window</td>
<td>144</td>
</tr>
<tr>
<td>5.10</td>
<td>The power demands of Google DC and the total power drawn from the grid by the full-horizon and the limited-horizon (Oracle) controllers for the LA battery</td>
<td>145</td>
</tr>
<tr>
<td>5.11</td>
<td>Google DC's expenses when the Full-Horizon controller is making workload shifting decisions for different Delay Window length</td>
<td>147</td>
</tr>
<tr>
<td>5.12</td>
<td>Google DC's total expenses when only the power demands for the low and non delay-sensitive jobs are allowed to be delayed</td>
<td>148</td>
</tr>
<tr>
<td>5.13</td>
<td>Breakdown of Google power consumption for the different job classes</td>
<td>149</td>
</tr>
<tr>
<td>5.14</td>
<td>Google DC's total expenses when our Full-Horizon controller is making only Workload Shifting, only Energy Storage and both Energy Storage &amp; Workload Shifting decisions</td>
<td>151</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.15</td>
<td>Google DC’s total expenses when our Limited-Horizon controller is making only Workload Shifting, only Energy Storage and both Energy Storage &amp; Workload Shifting decisions. Results are normalized with respect to ”No Peak Shaving” costs.</td>
<td>152</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Characteristics of Google Traces</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Configurations of the PMs within the Google cluster</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Energy measurements needed to calculate $T_{be}$</td>
<td>27</td>
</tr>
<tr>
<td>2.4</td>
<td>Avg. CPU and Memory Utilization over all testing data.</td>
<td>47</td>
</tr>
<tr>
<td>2.5</td>
<td>Total number of underestimated PMs during the entire testing period.</td>
<td>49</td>
</tr>
<tr>
<td>2.6</td>
<td>Execution time analysis of our framework’s stages.</td>
<td>50</td>
</tr>
<tr>
<td>3.1</td>
<td>Configurations of the PMs within the Google cluster</td>
<td>69</td>
</tr>
<tr>
<td>3.2</td>
<td>Experiment’s parameter values.</td>
<td>77</td>
</tr>
<tr>
<td>4.1</td>
<td>Evaluation of the proposed migration heuristic under different values of $N_{on}$ and $N_{sleep}$ during the entire testing period.</td>
<td>114</td>
</tr>
<tr>
<td>5.1</td>
<td>Specs for the considered battery types [1].</td>
<td>137</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>Modified BFD – estimating the number of needed PMs during the next prediction window</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>$[\theta, \gamma] = \text{Proposed Heuristic}(N_{on}, N_{sleep})$</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>LimitedHorizonControl($j, \bar{D}<em>j, \Phi, w, t</em>{max}$)</td>
<td>136</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

1.1 Why Worry About Energy?

Energy efficiency has become a major concern in large data centers. In the United States, data centers consumed about 1.5% of the total generated electricity per year, an amount that is equivalent to the annual energy consumption of 5.8 million households [2]. In US dollars, this translates into power costs of 4.5 billion per year. Data center owners, as a result, are eager now more than ever to save energy in any way they can in order to reduce their operating costs.

There are also increasing environmental concerns that too call for the reduction of the amounts of energy consumed by these large data centers, especially after reporting that the Information and Communication Technology (ICT) itself contributes about 2% to the global carbon emissions [3]. These energy costs and carbon footprints are expected to increase rapidly in the future as data centers are anticipated to grow significantly both in size and in numbers due to the increasing popularity of their offered services. All of these factors have alerted industry, academia, and government agencies to the importance of developing and coming up with effective solutions and techniques that can reduce energy consumption in data centers.

Cloud data centers are examples of such large data centers whose offered services are gaining higher and higher popularity, especially with the recently witnessed increasing reliance of mobile devices on cloud services [4–6]. The focus of this thesis is on energy consumption efficiency in cloud data centers.
1.2 The Cloud Paradigm

In the cloud paradigm, a cloud provider company owns a cloud center which consists of a large number of servers also called physical machines (PMs). These PMs are grouped into multiple management units called clusters, where each cluster manages and controls a large number of PMs, typically in the order of thousands. A cluster can be homogeneous in that all of its managed PMs are identical, or it could be heterogeneous in that it manages PMs with different resource capacities and capabilities.

Cloud providers offer these computing resources as a service for their clients and charge them based on their usage in a pay-as-you-go fashion. Cloud clients submit requests to the cloud provider, specifying the amount of resources that they need to perform certain tasks. Upon receiving a client request, the cloud provider scheduler creates a virtual machine (VM), allocates the requested resources to it, chooses one of the clusters to host the VM and assigns the VM to one of the PMs within that cluster. Client requests are thus also referred to as VM requests. After this allocation process takes place, the client can then use its allocated resources to perform its tasks. Throughout the VM lifetime, the cloud provider is expected, as well as committed, to guarantee and ensure a certain level of quality of service to the client. The allocated resources are released only once the client’s task completes.

1.3 New Challenges, New Opportunities

Energy efficiency has been a hot topic even before the existence of the cloud paradigm where the focus was on saving energy in laptops and mobile devices in order to extend their battery lifetimes [7–9]. Many energy saving techniques that were initially designed...
for this purpose were also adopted by the cloud servers in order to save energy. Dynamic voltage and frequency scaling and power gating are examples of such techniques. What is different in cloud centers is that we now have a huge number of servers that need to be managed efficiently. What makes this further challenging is the fact that cloud centers need to support on-demand, dynamic resource provisioning, where clients can, at any time, submit VM requests with various amounts of resources. It is this dynamic provisioning nature of computing resources that makes the cloud computing concept a great one. Such a flexibility in resource provisioning gives rise, however, to several new challenges in resource management, task scheduling, and energy consumption, just to name a few. Furthermore, the fact that cloud providers are committed to provide and ensure a certain quality of service to their clients requires extreme prudence when applying energy saving techniques, as they may degrade the quality of the offered service, thereby possibly violating Service Level Agreements (SLAs) between the clients and the cloud provider. Add on top of all of these challenges the fact that the electricity bill that the cloud data center receives at the end of the monthly billing cycle has a Peak Charge component which is basically a high penalty enforced by the grid company for the peak power demand that is drawn by the cloud data center within the monthly billing cycle. These high penalties should be avoided by balancing the amount of power that the cloud data center draws from the grid over the entire billing cycle which requires developing efficient scheduling and control strategies.

The good news after mentioning these challenges is that the adoption of the virtualization technology by the cloud paradigm brings many new opportunities for saving energy that are not present in non-virtualized environments. Examples of the opportunities brought by virtualization include the ability for multiple jobs to share the resources of the same server and the ability to migrate the scheduled jobs from a server into an-
other with minimal disruptions. Another good news is the fact that the cloud workload is heterogeneous where the hosted jobs have different resource elasticity, priority, delay tolerance, delay cost, lengths of execution time and generating revenue. This heterogeneity can reduce significantly the energy expenses and can increase the collected revenues if exploited correctly. Furthermore, cloud data centers are highly automated and are supplied with Uninterrupted Power Supply (UPS) batteries that can be used to balance the power drawn by the grid over the billing cycle which reduces the peak power demands and leads in turn into a significant monetary reduction in the cloud data center’s monthly electricity bill. Finally, the fact that cloud clusters are distributed across different geographic locations creates other resource management capabilities that can result in further energy savings if exploited properly and effectively.

1.4 Dissertation Contributions and Organization

The main contributions of this dissertation are the following. We:

- Propose a workload prediction and provisioning framework that predicts both the number of Virtual Machine (VM) requests that will be received in the future in addition to the amount of computing resources (e.g. CPU, memory) that are associated with each request (Chapter 2). These predictions are later used by our framework to estimate the number of servers in the cloud center that need to be kept ON to serve the future workload while the remaining servers are switched to sleep to save energy. The framework combines both k-means clustering techniques and adaptive Wiener filter predictions in order to forecast the future workload. Our framework also adapts its parameters over time and considers a dynamic safety margin in order to avoid underestimating the needed resources in future.
• Propose a resource allocation and scheduling framework that exploits the elasticity and price heterogeneity of the cloud tasks in order to maximize cloud profits while minimizing energy expenses (Chapter 3). This is done by reducing the duration during which servers need to be left ON and maximizing the monetary revenues when the charging cost for some of the elastic tasks depends on how fast these tasks complete, while meeting all resource requirements.

• Propose a resource overcommitment framework that predicts the future resource demands of scheduled VMs in the cloud data center, and uses these predictions to make efficient cloud resource overcommitment decisions with the objective of improving the utilization of the ON servers and reducing the cloud data center’s energy consumption (Chapter 4). The framework avoids server overloads by predicting them before occurring and by triggering appropriate VM migrations. The VM migration decisions to avoid overloads are made with two objective in mind: minimizing the number of ON servers and reducing the migration energy overhead.

• Propose a peak shaving framework that reduces the cloud data center’s peak power demands and hence the electricity bill through optimal energy storage and workload shifting control decisions (Chapter 5). The proposed framework operates under both full- and limited-knowledge of future power demands. Our framework also accounts for real energy storage losses and constraints, and considers a heterogeneous workload that is made up of jobs with different classes, with each class having different delay tolerance and price.

All of these frameworks are evaluated based on real traces from a Google cluster made up of more than 12K servers where we show the superiority of each of our frameworks when compared to the state-of-the-art cloud resource management techniques.

This chapter proposes an integrated energy-aware resource provisioning framework for cloud data centers. The proposed framework: i) predicts the number of virtual machine (VM) requests, to be arriving at cloud data centers in the near future, along with the amount of CPU and memory resources associated with each of these requests, ii) provides accurate estimations of the number of physical machines (PMs) that cloud data centers need in order to serve their clients, and iii) reduces energy consumption of cloud data centers by putting to sleep unneeded PMs. Our framework is evaluated using real Google traces collected over a 29-day period from a Google cluster containing over 12,500 PMs. These evaluations show that our proposed energy-aware resource provisioning framework makes substantial energy savings.

2.1 INTRODUCTION

Energy efficiency has become a major concern in large data centers. According to [2], U.S. data centers consumed about 1.5% of the total generated electricity in U.S. in 2006, an amount that is equivalent to the annual energy consumption of 5.8 million households. This consumption is also expected to increase as data centers are anticipated to grow both in size and in numbers. A recent study by Cisco predicts that cloud traffic will grow 12-fold by 2015 [10]. There are also increasing environmental concerns to reduce
energy consumption in industry after reporting that Information and Communication Technology (ICT) is responsible for about 2% of the global carbon emissions, equivalent to aviation [3]. All of these factors have alerted researchers to the importance of finding efficient solutions to save energy in data centers.

Cloud centers are examples of such large data centers. They often consist of a large number of servers also called physical machines (PMs). These PMs are grouped into multiple management units called clusters. Each cluster manages and controls a large number of PMs, typically in the order of thousands. A cluster can be homogeneous in that all of its managed PMs are identical, or it could be heterogeneous in that it manages PMs with different resource capacities.

Cloud providers offer these computing resources as a service for their clients and usually charge them based on their usage in a pay-as-you-go fashion. Cloud clients submit requests to the cloud provider, specifying the amount of resources they need to perform certain tasks. Upon receiving a client request, the cluster scheduler allocates the demanded resources to the client and assigns them to a PM. The virtualization technology allows the scheduler to assign multiple requests possibly coming from different clients to the same PM. Client requests are thus referred to as virtual machine (VM) requests. Cloud centers need then to support on-demand, dynamic resource provisioning, where clients can, at any time, submit VM requests specifying any amount of resources. It is this dynamic provisioning nature of computing resources that makes the cloud computing concept a great one [11]. Such a flexibility in resource provisioning gives rise, however, to several new challenges in resource management, task scheduling, and energy consumption, just to name a few [12, 13].

Energy consumption is of a special concern to cloud providers. According to a Google study [14], idle servers consume around 50% of their peak power. To save power, it is
therefore important to switch servers to the sleep mode when they are not in use. This requires the development of novel techniques that can monitor PMs and effectively decide whether and when they need to be put in sleep mode.

Towards the goal of creating an energy-aware cloud, we propose an integrated resource provisioning framework that i) predicts the number of future VM requests along with the amount of CPU and memory resources associated with each of these requests, ii) provides accurate estimations of the number of PMs that the cloud center needs in the future, and iii) makes intelligent power management decisions that reduce energy consumption. Our framework is evaluated using real Google traces [15] collected over a 29-day period from a Google cluster containing over 12,500 PMs.

Our proposed energy-aware resource provisioning framework is simple, adaptive, and effective in that it does not require heavy calculations, yet provides very accurate workload predictions and makes substantial energy savings in cloud centers. In addition, it requires minimum storage capacity for storing traces needed to train the developed models. To sum up, our main contributions in this work are as follows. We:

- develop a prediction approach that combines machine learning clustering and stochastic theory to predict both the number of VM requests and the amount of cloud resources associated with each request.

- propose adaptive enhancements to our predictor that make the prediction parameters tunable in realtime based on the actual request load. This increases the prediction accuracy over time and avoids the need for frequent model training that other machine learning approaches, such as Neural Network, suffer from.

- propose an integrated resource provisioning framework that relies on the proposed prediction approaches to make suitable energy-aware resource management deci-
• use real Google data traces to evaluate the effectiveness of our framework. These traces are collected from a heterogeneous Google cluster that contains more than 12K PMs.

The remainder of this chapter is organized as follows. Section 2.2 reviews the related work. Section 2.3 briefly describes the different components of our proposed framework. Section 2.4 presents the Google data used in this work, as well as the workload clustering approach. Section 2.5 presents the proposed prediction approach. Section 2.6 presents enhancements to the proposed prediction approach. Section 2.7 presents our power management heuristic. Section 2.8 evaluates the performance of our integrated resource provisioning framework. Finally Section 2.9 concludes the chapter and presents our future work.

2.2 RELATED WORK

Previous work on energy savings in cloud centers targeted two levels: i) server level, where the focus is on minimizing energy consumption of each server separately; and ii) data center level, where the focus is on efficiently orchestrating the pool of servers.

2.2.1 Server Level Techniques

Many energy efficient techniques initially designed to extend battery lifetime of laptops have also been applied to save energy in general-purpose servers. These techniques target reducing energy at all computer layers: hardware, OS, compiler, and application. Due
to space limitation, we only cover here OS-layer related approaches, as they involve some workload prediction relevant to our work. Readers interested in the work done on the other layers may refer to [9].

The basic idea behind OS-layer-based power management approaches is to save energy by switching idle devices, such as hard disk, to lower energy states whenever possible. The problem here is that because energy overhead due to switching the device to a lower or upper state is relatively high [16], it is worth switching a device to a lower state only when the device remains idle during a time period long enough to compensate for switching overhead. Researchers suggest first using a time-out approach [17], where a device is switched to an idle state only when it remains idle beyond a certain time-out threshold. A main disadvantage of such a simple approach is that it does not switch the device to a lower state until the time-out period has passed, resulting in not saving any energy during that period. To address this issue, researchers suggest then to use more advanced techniques based on machine learning [18, 19] to predict the idle time period length and rely on those predictions to make power management decisions.

Although many prediction approaches are proposed at the OS layer, they cannot be applied directly to predict VM requests. What these approaches predict is the length of the idle time whereas in cloud centers, multiple related parameters need to be predicted such as the number of arriving VM requests, the requested CPU and memory resources, etc. Furthermore, the proposed OS-layer related techniques study workload variations of a single device (e.g., hard disk) with the objective of saving energy in a single PM component. Unlike these techniques, we study workload variations of a cloud cluster with the goal of turning an entire PM to a lower state. It is worth mentioning that OS power management techniques are complementary to our work, as they can be applied on top of our framework to manage the different components of the active PMs.
2.2.2 Data Center Level Techniques

Energy awareness in data centers have focused on two aspects: VM consolidation and cluster-level power management.

2.2.2.1 VM Consolidation

Researchers investigated first where to place the received VM requests within the cloud cluster using the least number of ON PMs. This problem is similar to the standard Bin Packing (BP) optimization problem, which views VMs as objects and PMs as bins and where the objective is to pack these objects in as few bins as possible. The problem is known to be NP-hard [20], and thus approximation heuristics, such as First Fit Decreasing (FFD) and Best Fit Decreasing (BFD) have been proposed [21, 22] instead. Although these heuristics provide very close approximations to the optimal placement solution, they assume that all coming VM requests are known ahead of time, which is not the case for the on-demand computing paradigm. When requests are not known ahead of time, online BP heuristics, such as First Fit, Next Fit, and Best Fit have been proposed [23–25] to be used instead. The main problem with those online heuristics is that they don’t anticipate the future workload and make VM hosting and PM switching decisions based only on the currently received VM requests. Unlike those heuristics, our framework takes a further step by predicting the VM requests to be received in future and relies on that to make intelligent power management decisions.

Given that VM requests can be terminated any time the cloud client wants, it is then highly likely that PMs become under-utilized over time, resulting in inefficient use of data center resources. Migration and dynamic consolidation techniques have been
proposed as key solution approaches for improving datacenter resource efficiency [5, 26–30]. For example, dynamic consolidation approaches proposed in [26–29] allow VMs to migrate from the under-utilized PMs so that the workload can be consolidated on a smaller subset of PMs, allowing further PMs to be turned to sleep. The authors in [26, 27] considered the performance degradation associated with the migration process and proposed a dynamic consolidation framework that guarantees certain SLA level. The work in [30] took the migration energy overhead into account when making such decisions. All the mentioned dynamic consolidation techniques are complimentary to our work as they can be applied on top of our framework to pack the already scheduled VMs more tightly over time. Our work is different as we are predicting the number and the amount of VM requests to be received in the future. In fact, these predictions can improve the migration decisions made by those dynamic consolidation techniques by considering not only the current hosted requests, but also the future requests predicted by our framework when making their migration decisions.

Figure 2.1: Flow chart of the proposed framework
2.2.2.2 Cluster-Level Power Management

The work in [31, 32] exploited the fact that servers in cloud centers are distributed in different geographical locations with varying electricity prices and proposed different strategies for placing the received requests on these distributed servers such that the energy costs are minimized. Although these techniques reduce the cluster’s electricity bills, they do not actually reduce the consumed energy.

Power management approaches that save energy by adjusting CPU’s operating frequency and voltage of PMs based on their workload within the cluster have also been proposed [33, 34] (these are known as Dynamic Voltage and Frequency Scaling (DVFS)). Unlike these techniques, in our framework redundant PMs are completely turned to sleep mode instead of reducing their operating frequency and voltage, thereby achieving higher energy savings.

Several predictive frameworks were proposed in [35–40] to reduce the number of ON PMs by tuning the allocated resources of the already scheduled VMs. The work in [35–37] considers the case where a client requests multiple VMs to run a certain application. Rather than reserving a fixed number of VMs for each application all the time, the authors dynamically adjust this number based on predicting the application’s demands. PRESS [38] on the other hand performs VM resizing where the allocated resources (e.g. CPU or memory) for each scheduled VM are tuned based on predicting the client’s demands. The authors in [39, 40] add dynamic consolidation on top of VM resizing and thus use VM migration to compact the resized VMs on fewer ON PMs. Our framework is different from those predictive frameworks as we are predicting the number of VM requests to be received from all the cloud clients in addition to the resource demands associated with each request. These predictions are used later to estimate the number
of ON PMs needed to support the future workload. This is different from predicting the number of needed VMs for a specific application that was already scheduled on the cluster [35–37], or the future resource demands of an already scheduled VM [38–40].

There have been many approaches proposed to predict the coming load in distributed systems, such as data centers and grid systems. Hidden Markov models [41] and polynomial fitting [42] are examples of workload prediction methods that have been used in Grid systems. The fact that cloud workloads are made up of requests with multiple resource demands makes such predictions schemes not applicable to the cloud paradigm.

2.3 THE PROPOSED FRAMEWORK

Our framework has three major components: data clustering, workload prediction, and power management. In this section, we briefly describe these components so as the reader will have a global picture of the entire framework before delving into the details. Detailed descriptions are provided in later sections. Throughout this section, we refer to Fig. 2.1 for illustration.

2.3.1 Data Clustering

Our developed prediction approach relies on observing and monitoring past workload variations during a time period referred to as the observation window in order to predict the workload coming in a certain future period referred to as the prediction window. A VM request requested by a cloud client typically consists of multiple cloud resources (e.g., CPU, memory, bandwidth, etc.). This multi-resource nature of the requests poses unique challenges when it comes to developing prediction techniques. Also, different
cloud clients may request different amounts of the same resource. Therefore, it is both impractical and too difficult to predict the demand of each type of resource separately, as ideally this is what is needed to be able to make optimal power management decisions. To address this issue, we instead divide VM requests into several categories, and then develop prediction techniques for each of these categories. This is known as clustering.

2.3.1.1 k-Means clustering

Our first step is then to create a set of clusters to group all types of VM requests; i.e., each VM request is mapped into one and only one cluster, and all requests belonging to the same cluster possess similar characteristics in terms of their requested resources.

In the general case, each VM request is associated with \( d \) types of resources such as CPU, memory, bandwidth, etc. In order to divide these requests into multiple categories, we represent first each request as a point in the \( \mathbb{R}^d \) space. As for the Google cluster [15], only two types of resources, CPU and memory, are associated with each request. Thus these requests are represented in the \( \mathbb{R}^2 \) space, where each point is a request and the two dimensional coordinates of the point are the amount of CPU and memory resources associated with the request. Clustering these data points into a number of clusters is done using k-Means algorithm [43].

As shown in Fig. 2.1, the k-Means algorithm takes as an input the Google traces and the number of clusters, \( k \), and outputs \( k \) clusters, each specified by its center point. For the case of the Google data, where requests have two types of resources (CPU and memory), the cluster centers are points in the \( \mathbb{R}^2 \) space. In the general case, when requests have \( d \) types of resources, the cluster centers would be points in the \( \mathbb{R}^d \) space. For illustration purposes only, Fig. 2.1 shows an example with \( k = 2 \), where two clusters
produced by the algorithm are represented in Red and Blue, as shown in the Cluster Groups graph. Note that the parameter $k$ needs to be chosen \textit{a priori} and given as an input to the clustering algorithm. In Section 2.4.2, we show how such a parameter is selected.

It is worth mentioning that the clustering stage is done only once on large traces collected from the cloud center. The clustering stage can be launched again if the request characteristics change significantly in the cloud center over time. Our experiments on the Google traces show that running the clustering phase only once on a large training data is enough to extract the characteristics of the submitted VM requests.

2.3.1.2 Traces Decomposer

Once the $k$ clusters and their center points are determined, they are given as an input to the Traces Decomposer module (shown in Fig. 2.1), which is responsible for mapping each request received during the observation window into one cluster. The observation window is split into $L + 1$ time slots, $n, n - 1, \ldots, n - L$, as follows. Suppose a prediction needs to be made at time $n$. In this case, slot $n$ corresponds to time interval $[n - 10, n]$ (in seconds); slot $n - 1$ corresponds to time interval $[n - 20, n - 10]$, slot $n - i$ corresponds to time interval $[n - 10(i + 1), n - 10i]$, and so on. The Traces Decomposer tracks the number $x_{n-i}$ of received requests in time slot $[n - 10(i + 1), n - 10i]$ of the observation window for all $i = 0, 1, \ldots, L$, and maps each request within the slots into one cluster.
2.3.2 Workload Prediction

We use stochastic Wiener filter prediction to estimate the workload of each category/cluster. The Stochastic Predictor, as shown in Fig. 2.1, is made of $k$ Wiener filters. Each filter takes as an input the number of received requests for a certain category during the observation window, and uses it to predict the number of requests of that same category to arrive in the prediction window. This makes the problem easier to solve as there are infinite number of possibilities for the amount of CPU and memory resources that a client may request.

The reasons for choosing Wiener filter as a predictor are: First, it outperforms the other schemes in terms of prediction accuracy as will be seen in Section 2.6.2. Second, it is simple, as the prediction for each category is a weighted summation of the recently observed number of requests of that category. Third, it has a sound theoretical basis. Finally, it is easy to improve the original Wiener filter to support online learning, making it more adaptive to changes in workload characteristics. This is done by updating the filter’s weights as new observations are made over time without requiring heavy calculations or large storage space as will be seen in Section 2.6.1.

The parameters that need to be determined for each filter branch are: the length of the observation window, the length of the prediction window, and the weights. These parameters are determined in Section 2.5.

2.3.3 Power Management

The predictions of all categories along with their center points are all next passed to the Power Management module, which uses this information to decide which PMs need
to go to sleep and which ones need to be kept ON. This unit keeps track of all PMs in the cloud cluster and stores their current utilizations and states (ON or sleep). It uses a modified Best Fit Decreasing (BFD) heuristic to fit the predicted VM requests in PMs in order to determine how many ON PMs will be needed in the coming prediction window.

The original BFD algorithm [22] tries to pack VM requests in the fullest PM with enough space. In order to do that, it sorts PMs from the fullest to the least full and iterates over the ordered list of PMs trying to pack the VM request within the first PM that has enough space. The original algorithm could not be applied directly in our framework as it considers only one dimension and thus a modification is needed to make the algorithm work for the case where PMs and VMs have multiple dimensions (i.e. multiple types of resources). This limitation has been addressed by mapping these multiple dimensions into a single metric that combines them all. Examples of such a metric include taking the sum or the product of those multiple dimensions. In our heuristic, we considered the product metric, as our experiments show that it outperforms the summation metric. Furthermore, our proposed heuristic takes the energy efficiency problem into account when sorting the PMs as it sorts PMs by the following criteria (in ascending order):

(i) PMs that are ON

(ii) PMs that have higher utilizations. The utilization metric is defined as the product of the CPU and the memory capacities of the PM.

(iii) PMs that have higher capacities. Similarly, the capacity metric is defined as the product of the memory and CPU capacities of the PM.

\(^1\)Other combining metrics will be investigated in the future.
The intuition behind our sorting criteria is as follows: we want to make use of the available ON PMs, which already have some scheduled VMs; so ON PMs are ranked first. We then use the utilization metric as the next sorting criterion, since increasing the utilization of the PMs makes the cluster as a whole more energy-efficient, as it results in switching to sleep more PMs. Finally, PMs are sorted based on their capacities, as one can fit more VMs within a PM when PMs are of large capacities. Detailed description of the proposed energy saving heuristic is provided in Section 2.7.

2.4 Data Clustering

In this section, we first begin by presenting the Google data traces that we used in this work to train and test our developed energy-aware resource provisioning models, and then present our workload clustering and classification findings.

2.4.1 Google Traces

We conduct our experiments on real Google data [15] that was released in November 2011 and consists of a 29-day traces collected from a cluster that contains more than 12,500 PMs. A summary of this data is provided in Table 2.1.

The cluster is heterogeneous as it supports different PM configurations, as described in Table 2.2. The first column shows the number of PMs in the cluster whose configurations are specified in the subsequent columns. The second column shows the architecture type of these PMs. Note that there are three different types of architecture, referred to as A, B and C, as their actual type has been obfuscated for privacy reasons. PMs from the same architecture may have different CPU and memory capacities as shown in column
three and four, respectively. These capacities are normalized to the maximum capacity also for privacy reasons, and thus the reported capacities are all less than or equal to one.

The traces provided by Google are collected at the cluster level, where VM requests are submitted and scheduled, and at the PM level too, where the amount of utilized resources are tracked over time for each VM. Previous work on this data has focused on studying general statistical characteristics of the cloud cluster [44, 45] or on classifying VMs into a number of groups based on the amount of resources they utilize over time [46].

Table 2.1: Characteristics of Google Traces

<table>
<thead>
<tr>
<th>Trace Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration of Traces</td>
<td>29 days</td>
</tr>
<tr>
<td>Number of PMs</td>
<td>12,583</td>
</tr>
<tr>
<td>Number of VM requests</td>
<td>$\geq$ 50M</td>
</tr>
<tr>
<td>Compressed size of data</td>
<td>39GB</td>
</tr>
</tbody>
</table>

Table 2.2: Configurations of the PMs within the Google cluster

<table>
<thead>
<tr>
<th>Number of PMs</th>
<th>PM Configurations</th>
<th>Architecture</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>6732</td>
<td>A</td>
<td>0.50</td>
<td>0.50</td>
<td></td>
</tr>
<tr>
<td>3863</td>
<td>A</td>
<td>0.50</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>A</td>
<td>0.50</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>795</td>
<td>C</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>B</td>
<td>0.25</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>A</td>
<td>0.50</td>
<td>0.12</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td>0.50</td>
<td>0.03</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td>0.50</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>1.00</td>
<td>0.50</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>0.50</td>
<td>0.06</td>
<td></td>
</tr>
</tbody>
</table>

The clustering step implemented in our framework is different from the work in [46], as VM requests are clustered into multiple categories based on the amount of requested CPU and memory, rather than classifying these requests based on the amount of resources they utilize over time. To the best of our knowledge, this is the first work that considers
developing a workload prediction approach based on this data.

Our experiments utilize the data provided in the task event table, where each VM request is called a task and each VM submission/termination request is referred to as an event. It is worth mentioning that Google chooses to allocate containers rather than full virtual machines for these tasks. Each one of these two choices has their advantages and disadvantages. Full machine virtualization offers greater isolation at the cost of greater overhead, as each virtual machine runs its own full kernel and operating system instance. Containers, on the other hand, generally offer less isolation but lower overhead through sharing certain portions of the host kernel and operating system instance. As far as our framework is concerned, we are predicting the future workload by predicting the number of task requests and amount of resources associated with these requests. These tasks could be handled by either full virtualization or by containers but this does not affect the applicability of our framework in predicting the future workload and in estimating the amount of needed PMs for the future workload. Throughout the paper, we refer to these tasks as VMs. A detailed description of the data is provided in [47], and the following are the features used by our framework:

- **VM ID**: a unique identifier for each VM. Kept anonymous and replaced by its hash value.

- **Client ID**: a unique identifier for each cloud client. Kept anonymous and replaced by its hash value.

- **Event type**: specifies whether the event is a submission or a release request. It is worth noting that clients may submit or release a VM request whenever they desire.
- Timestamp: time at which the event happened.
- Requested CPU: amount of requested CPU.
- Requested memory: amount of requested memory.

Note that the requested amount of CPU (memory) resources is always between 0 and 1 as it is normalized to the PM with the largest CPU (memory) capacity in the Google cluster. Also note that clients do not necessarily use all requested resources at all the time, as their usage varies depending on their needs. Yet these resources are reserved and allocated for them for so long as their requests are not released, regardless of whether they are using them or not.

Since the size of the Google data is huge, we use two chunks of the traces to tune the different parameters that are involved in our framework. We refer to these chunks as the training set and the validation set. The training set includes the Google traces collected during a 24-hour period and the validation data set contains the traces collected during a 16-hour period. A separate chunk of the Google data called the testing set with a duration of 29 hours is used later to estimate the accuracy, the energy savings and the utilization that our framework achieves. The testing chunk is only used to evaluate our framework’s performance on unseen data and no parameters are selected based on this chunk.

2.4.2 Workload Clustering

We use k-Means [43], a well-known unsupervised learning algorithm, to cluster VM requests into $k$ categories. The k-Means algorithm assigns $N$ data points to $k$ different clusters, where $k$ is a priori specified parameter. The algorithm starts by an initializa-
tion step where the centers of the $k$ clusters are chosen randomly, and then assigns each data point to the cluster with the nearest (according to some distance measure) center. Next, these cluster centers are recalculated based on the current assignment. The algorithm repeats i) assigning points to the closest, newly calculated clusters and then ii) recalculating the new centers until the algorithm converges (no further changes occur).

As mentioned previously, each VM request is mapped into $\mathbb{R}^2$ where the two dimensions are the requested amount of CPU and memory. The Euclidean distance is used as the measure of distance between the points and the centers. k-Means is run for 20 different initial centers and the resulting clusters with the lowest error are reported.

One of the important parameters that needs to be tuned when using k-Means is $k$, the number of clusters. A heuristic approach is implemented to tune this parameter, in which the Sum of Squared Distances (SSD) is plotted as a function of the number of clusters $k$. SSD represents the clustering error when each point in the data set is represented by its corresponding cluster center, and is mathematically equal to $\sum_{i=1}^{k} \sum_{r \in C_i} d(r, c_i)^2$ where $C_i$ denotes cluster $i$; i.e., set of all points belonging to the $i^{th}$ cluster, $c_i$ denotes cluster $i$’s center point, and $d(r, c_i)$ is the Euclidean distance between $r$ and $c_i$.

Fig. 2.2 shows $SSD$ as a function of the number of clusters $k$ plotted based on the
training set of the Google traces. Note that as $k$ increases, $SSD$ decreases monotonically, and hence so does the clustering error. Recall that while increasing the value of $k$ reduces the clustering error, it also increases the overhead incurred by the prediction technique (to be presented in next sections), since a predictor branch needs to be built for each cluster/category. For this, the heuristic approach searches then for the "elbow" or "knee" point of the plot, which is basically the point that balances between these two conflicting objectives: reducing clustering errors and maintaining low overhead. As can be seen from Fig. 2.2 which is based on the training traces, the value 4 for $k$ strikes a good balance between accuracy and overhead. Hence, in what follows we use $k = 4$.

Fig. 2.3 shows the resulting clusters for $k = 4$ based on the training set, where each category is marked by a different color/shape and the centers of these clusters $c_1$, $c_2$, $c_3$ and $c_4$ are marked by ‘x’. Category 1 represents VM requests with small amount of CPU and small amount of memory; Category 2 represents VM requests with medium amount of CPU and small amount of memory; Category 3 represents VM requests with large amount of memory (and any amount of requested CPU). Category 4 represents VM requests with large amount of CPU (and any amount of requested memory). Observe from the obtained clusters that requests with smaller amount of CPU and memory are
denser than those with large amounts.

2.5 Workload Prediction

In this section, we determine and estimate the parameters of the proposed prediction approach.

2.5.1 Length of the Prediction Window

An important parameter that needs to be estimated for the stochastic predictor is the length of the prediction window, $T_p$. This represents the length of the time period in the future for which the workload needs to be predicted in order to decide whether or not PMs need to be switched to sleep mode. Letting $P_{idle}$ denote the power the PM consumes while in ON and idle, the amount of energy that the PM consumes if it is left ON and idle during $T_p$ is $E_{idle} = P_{idle} \times T_p$. If we decide to switch the PM to the sleep mode, then the consumed energy $E_{sleep} = E_o + P_{sleep} \times (T_p - T_o)$, where $P_{sleep}$ is the consumed power when in the sleep mode; $E_o$ is the transition energy, equaling the energy needed to switch the machine to the sleep mode plus the energy needed to wake up the machine later; and $T_o$ is the transitional switching time.

Let $T_{be}$ be the amount of time during which keeping the PM ON and idle consumes an amount of energy that is equal to the energy consumed due to mode transition plus that consumed while the PM is in the sleep mode during that same period. Here, $T_{be}$ represents the break-even time, and must satisfy:

$$P_{idle} \times T_{be} = E_o + P_{sleep} \times (T_{be} - T_o)$$
Note that switching a PM to sleep mode saves energy only when the PM stays idle for a time period longer than $T_{be}$; that is, $T_p \geq T_{be}$ must hold in order for the power switching decisions to be energy efficient.

In this work, we rely on the energy measurement study of physical machines conducted in [48] to estimate the break-even time, $T_{be}$. Table 2.3 shows those measurements (reported in [48]) that are used to calculate $T_{be}$. These measurements are also used later to estimate the energy savings our proposed techniques achieve. Based on these measurements, our calculation yields $T_{be} = 47$ seconds; that is, $T_p$ should be larger than 47 seconds. It is worth noting that these energy measurements were based on a certain type of commercial PMs, and hence, these numbers might slightly change depending on the type of PMs in the cluster. As mentioned before, Google does not provide information about the types of these PMs. Although we took in our experiments a conservative approach and chose $T_p = 60$ seconds in order to account for the cases where these numbers might be different for other types of PMs, our proposed approach works for any other $T_p$ choice.

It is worth mentioning that since $T_p = 60$, the predictors in our framework are run every minute to predict the number of requests in the coming minute. The power management module relies on these predictions to estimate the number of needed PMs in the next minute. Since Switching PMs from sleep to ON takes time, then the predictors are run before the minute ends by an amount of time that is equal to the switching time so as to have the PMs ready to cover the workload in the coming minute.
Table 2.3: Energy measurements needed to calculate $T_{be}$

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{sleep}$</td>
<td>107</td>
<td>Watt</td>
</tr>
<tr>
<td>$P_{idle}$</td>
<td>300.81</td>
<td>Watt</td>
</tr>
<tr>
<td>$P_{peak}$</td>
<td>600</td>
<td>Watt</td>
</tr>
<tr>
<td>$E_{on\rightarrow sleep}$</td>
<td>5510</td>
<td>Joule</td>
</tr>
<tr>
<td>$E_{sleep\rightarrow on}$</td>
<td>4260</td>
<td>Joule</td>
</tr>
<tr>
<td>$T_o$</td>
<td>6</td>
<td>Seconds</td>
</tr>
</tbody>
</table>

Figure 2.4: The general structure of Wiener filter.

2.5.2 Weights of the Stochastic Predictor

Let $n$ be the time at which the prediction needs to be made. The general structure of the Wiener predictor for each of the four categories can be represented as shown in Fig. 2.4, where:

- $x_{n-i}$: is the number of requests of the considered category received in the period between $n - 10(i + 1)$ and $n - 10i$ seconds.

- $d_n$: the desired output of the category predictor. This represents the actual number of requests for the considered category in the coming prediction window.

- $\hat{d}_n$: is the predicted number of requests for the considered category in the coming prediction window.

- $L$: is the number of taps that the predictor relies on in making predictions.

- $w_i$: is the $i^{th}$ tap's weight.
Wiener filter predicts the future requests assuming $x_n$ is a wide-sense stationary process. The predicted number of requests, $\hat{d}_n$, is a weighted average of the previous observed requests and thus can be written as $\hat{d}_n = \sum_{i=0}^{L} w_i x_{n-i}$. The prediction error, $e_n$, can be calculated as the difference between the actual and predicted number of requests; i.e.,

$$e_n = d_n - \hat{d}_n = d_n - \sum_{i=0}^{L} w_i x_{n-i}$$

The objective is to find the weights that minimize the Mean Squared Error ($MSE$) of the training data, which is:

$$MSE = E[e_n^2]$$  \hspace{1cm} (2.1)

$MSE$ represents the second moment of error, and the reason we chose it as an objective function is due to the fact that it minimizes both the average error and the variance of error putting more weight on the average error ($E[e_n^2] = (E[e_n])^2 + var(e_n)$).

Differentiating Eq. (2.1) with respect to $w_k$ and setting this derivative to zero yields, after some algebraic simplifications,

$$E[d_n x_{n-k}] - \sum_{i=0}^{L} w_i E[x_{n-k} x_{n-i}] = 0$$

By letting

$$R_{dx}(k) = E[d_n x_{n-k}]$$  \hspace{1cm} (2.2)

$$R_{xx}(i-k) = E[x_{n-k} x_{n-i}]$$  \hspace{1cm} (2.3)

it follows that $R_{dx}(k) = \sum_{i=0}^{L} w_i R_{xx}(i-k)$. 
Similar equations expressing the other weights can also be obtained in the same way. These equations can be presented in a matrix format as \( R_{dx} = R_{xx}W \), where

\[
R_{xx} = \begin{bmatrix}
R_{xx}(0) & R_{xx}(1) & \ldots & R_{xx}(L) \\
R_{xx}(1) & R_{xx}(0) & \ldots & R_{xx}(L-1) \\
\vdots & \vdots & \ddots & \vdots \\
R_{xx}(L) & R_{xx}(L-1) & \ldots & R_{xx}(0)
\end{bmatrix}
\]

\[
W = \begin{bmatrix} w_0 & w_1 & \ldots & w_L \end{bmatrix}^T
\]

\[
R_{dx} = \begin{bmatrix} R_{dx}(0) & R_{dx}(1) & \ldots & R_{dx}(L) \end{bmatrix}^T
\]

Given \( R_{xx} \) and \( R_{dx} \), the weights can then be calculated as:

\[
W = R_{xx}^{-1}R_{dx} \tag{2.4}
\]

We rely on the training data set to calculate \( R_{xx} \) and \( R_{dx} \) for each category. To estimate these parameters for a certain category, we divide the training data into \( N \) slots where the duration of each slot is 10 seconds. We first calculate the number of requests of the considered category that are received in each slot. Then, we calculate the elements of \( R_{xx} \) using the unbiased correlation estimation as:

\[
R_{xx}(m) = \frac{1}{N-m} \sum_{j=0}^{N-m-1} x_{j+m}x_j \tag{2.5}
\]

The elements of \( R_{dx} \) can also be estimated using the correlation coefficients. Since \( d_n \) represents the number of requests in the coming prediction window which has a duration
of 60 seconds, we can write \( d_n = \sum_{i=1}^{6} x_{n+i} \). Plugging the expression of \( d_n \) in Eq. (2.2) yields the correlations that can be estimated from the training data. An estimation of the weight vector follows then for each category predictor provided \( R_{dx} \) and \( R_{xx} \) are known. These weights lead, in turn, to the lowest MSE for the training data.

2.5.3 Length of the Observation Window

The last parameter that needs to be tuned for each category predictor is the length of the observation window. As mentioned before, the observation window is divided into \( L \) slots or (also called) taps, each tap/slot is of length 10 seconds. This Wiener filter is referred to as an \( L \)-tapped filter. To determine \( L \) for a category predictor, we first need to find the optimal weight vectors of the Wiener predictor under different values of \( L \) on the training data, and then test the performance of these weight vectors on the unseen validation data.

Fig. 2.5 shows the Root Mean Square Error (RMSE) of the 3rd category predictor for both the training and validation data sets for different values of \( L \). Observe that
the training data $RMSE$ decreases monotonically as $L$ increases. To understand this, consider two Wiener filters: one with $L$ taps and the other with $L + R$ taps. Recall that we need to find optimal weights that when multiplied by these taps leads to the minimum $MSE$ of the training data. As a result, the model with $L + R$ taps can achieve the same accuracy on the training data set as the model with $L$ taps by setting the weights of all the additional $R$ taps to zero. Thus, the model with a given number of taps will, in the worst-case scenario, achieve the same accuracy as any model with lower numbers of taps. In general, models with larger numbers of taps can still find some correlations specific to the training data that lead to a better accuracy. Consequently, the training error will continue to decrease as we increase the number of taps.

However, by observing the behavior of the validation data, note that $RMSE$ decreases first until it reaches a point beyond which the error can no longer be reduced even if $L$ is increased further. Also, observe that if we continue to increase $L$, the validation $RMSE$ starts to increase. This behavior is expected and is known as the overfitting phenomena. After increasing the number of taps beyond a certain limit, the model tries to find correlations between the different requests over time. These correlations are specific to the training data so we say that increasing the number of taps beyond a certain limit increases the complexity of the model and starts finding correlations that do not exist in the general traces but are specific to the training data. Based on our experiments, we chose $L = 80$, as it achieves the best accuracy on the unseen validation data, meaning that the observation window of the 3rd category predictor relies on the traces in the previous 80 taps collected in the previous $80 \times 10 = 800$ seconds.

Using a similar approach, the optimal numbers of taps for the other three categories are determined to be 20 (category 1), 80 (category 2), and 34 (category 4). Graphs for these three categories are omitted due to space limitation.
2.6 Prediction Enhancements

We now describe two enhancements that can be done on the proposed stochastic predictor. These enhancements aim at improving prediction accuracy and making more efficient cloud resource allocation and management.

2.6.1 Adaptive Predictor

In our framework, we rely on the traces from the training data to calculate the weights. One of the problems that might be encountered in a practical implementation of the framework is that we will need to retrain the model to adjust these weights since the workload characteristics may vary over time. We refer to such a model that needs training from time to time as the static Wiener model. We provide in this subsection an adaptive Wiener model that increases the accuracy over time and avoids the need to store large traces to retrain the model on new training data in order to update weights.

Initially, the adaptive model predicts the number of requests in the coming minute based on the learned weights from the training stage. Next, the model observes the number of requests that were received in that minute for each category. It uses these observations to update the weights. The adaptive model continues doing this every minute. As a result, our adaptive approach uses all the traces that were observed until the time of the prediction to find the optimal weights. This is different from the static approach that relies only on a chunk of the traces from the training data to adjust these weights. The adaptive predictor has an overhead since after observing the actual workload in each minute it needs to do some calculations to update these weights. However, these updates increase the accuracy over time and makes the model dynamic to the latest variations.
of the workload.

As showed previously in Section 2.5.2, we only need to calculate the correlations for different lags of time in order to calculate the weights. In order to reduce the calculation overhead in our adaptive model, we introduce two variables for each correlation coefficient that aggregate all the previous calculations. We only need to store these variables instead of storing all the previous traces and thus the amount of storage needed to update these weights is reduced. Furthermore, these variables can be updated easily once we observe the new workload which reduces the calculation overhead further.

The unbiased estimation of the coefficients given in Eq. (2.5) can be written as $R_{xx}(m) = \text{Sum}(m)/\text{Counter}(m)$, where $\text{Sum}(m) = \sum_{j=0}^{N-m-1} x_{j+m} x_j$ and $\text{Counter}(m) = N - m$ are two aggregate variables.

Recall that $x$ is a random variable that represents the number of requests received within ten seconds. In each minute, six new observations of $x$ take place. We refer to these new observations by the group $O$. The two aggregate variables, $\text{Sum}(m)$ and $\text{Counter}(m)$, are then updated as:

$$\text{Counter}(m) \leftarrow \text{Counter}(m) + 6$$

$$\text{Sum}(m) \leftarrow \text{Sum}(m) + \sum_{k \in O} x_{k-m} x_k$$

The new correlations can be estimated easily by dividing $\text{Sum}$ by $\text{Counter}$ for the different lags. Next, we calculate the updated weights using Eq. (2.4).

Fig. 2.6 shows the actual number of requests received for the third category over time. We also plot the predicted number of requests based on both the static and adaptive Wiener predictors. Note that initially both adaptive and static models give the same predictions as the adaptive model still has not made many updates. Observe how
the adaptive model improves over time as its predictions become closer to the actual number of requests. We evaluate the accuracy of both of these models in Section 2.6.2, and present the second enhancement in Section 2.6.3.

2.6.2 Prediction Accuracy

We now compare the accuracy of the two proposed adaptive and static Wiener predictors against one another and against six other prediction techniques, which are described next:

- **Last minute predictor**: returns the same number of requests observed during the previous minute.

- **Min predictor**: observes the number of requests received in each minute during the last five minutes, and returns the minimum of these five observations.

- **Max predictor**: similar to Min predictor but returns the maximum instead of the minimum.

- **Average predictor**: similar to Min predictor but returns the average instead of the minimum.
• **Exponential Weighted Moving Average (EWMA) predictor:** assigns exponentially decreasing weights for the previous observations rather than equal weights as in the Average Predictor. The smoothing factor for the EWMA predictor is tuned by evaluating the predictor on the training data using different smoothing factors. The value of the smoothing factor that achieves the highest accuracy on the training data is selected for future predictions.

• **Linear Regression (LR) predictor:** observes the number of requests received in each minute during the last five minutes, and returns a weighted average of the previous observations where the considered weights are tuned based on the training data.

Fig. 2.7 shows the Root Mean Square Error (RMSE) for the different predictive approaches. Since in each minute we need to predict the number of requests for four categories, the reported RMSE in Fig. 2.7 is the summation of the RMSE for the predictions of the four categories. These evaluations are reported on a testing data of Google Traces that includes 2.5 million VM requests received during a 29-hour period. The testing data set is different from the training and validation data sets that are used for tuning the parameters. This provides a fair comparison as it shows the performance of our predictor over new data that it did not see before.

Note also from Fig. 2.7 that both the static and adaptive Wiener predictors have lower RMSE than all the other predictive approaches. The adaptive Wiener predictor achieves lower RMSE than the static predictor as it updates the weights regularly based on the observations that are seen in each minute. This increases the accuracy of the predictor and makes it more adaptive to new workload changes, something that the other studied prediction techniques lack. The static Wiener predictor would have achieved an
accuracy that is close to that achieved by the adaptive predictor if it was trained regularly every certain duration of time such as every two hours. It is also worth mentioning that the basic predictive techniques have also been tested for different time duration (not only 5 minutes but 10 minutes, 15 minutes,...etc.). For all of these cases, the basic approaches report a performance worse than the static and adaptive Wiener approaches. We report only the basic predictive approaches that rely on the number of requests received during the last 5 minute duration in their prediction due to space limitation.

2.6.3 Safety Margin

Our stochastic predictors may still make errors as the predictions may be larger or smaller than the actual number of requests. A main problem when we underestimate the number of future requests is that we will need some time to wake one of the sleeping machines when an unpredicted request arrives. As a result, clients may observe an undesired short delay before their resources are allocated. In order to reduce the occurrences of such cases, we add a safety margin to accommodate for such variations. The cost of this
safety margin is that we will keep some redundant PMs in the working state which will reduce slightly the total energy savings.

In order to avoid the problem of selecting an appropriate static value for the safety margin, we make it dynamic and related to the accuracy of our predictors. Our proposed safety margin increases if our predictions deviate much from the actual number of requests and it decreases when accurate predictions are made. Since these deviations may vary over time, we calculate an exponentially weighted moving average (EWMA) of the deviations while giving higher weights to the most recent ones. Initially, $Dev_0 = 0$, and for later time $n$, $Dev_n$ is updated as

$$Dev_n = (1 - \alpha)Dev_{n-1} + \alpha|d_{n-1} - \hat{d}_{n-1}|$$

where $0 < \alpha < 1$ is the EWMA smoothing factor used to tune between the weight given to most recent deviations over that given to the previous ones. $Dev$ is updated before the next prediction is made by observing the deviation between the predicted and actual number of requests in the previous minute. One advantage of EWMA is that the moving average is calculated without needing to store all observed deviations.

We add the weighted average $Dev$ to our predictions in the next minute after we multiply it by a certain parameter $\beta$. The predicted number of requests with safety margin $\bar{d}_n$ can then be calculated as $\bar{d}_n = \hat{d}_n + \beta Dev_n$.

Figs. 2.8 and 2.9 show the actual number of requests in addition to the predictions with and without safety margin for the second and third categories on a part of the testing set. Note that the predictions for the remaining two categories were not included due to space limitation. As for the safety margin parameters, $\alpha$ and $\beta$ are set to 0.25 and 4, respectively; these values are selected experimentally by picking the values that
Figure 2.8: Adaptive prediction with and without safety margin for category 2.

Figure 2.9: Adaptive prediction with and without safety margin for category 3.
provide the best predictions on the training set. Note from these figures that the prediction with safety margin forms an envelope above the actual number of requests. The envelope becomes tighter when our predictions are accurate and becomes wider when our predictions deviate much from the actual number of requests for each category. When bursts in the actual number of requests occur, the envelope becomes wider directly after that burst to avoid any future possible bursts. If no other bursts are observed in the following minutes, then the envelope becomes tighter over time as the recent deviations which have higher weight are small.

We also show in Fig. 2.10 the total number of unpredicted PMs that were waken from sleep to accommodate the workload when our framework is evaluated on the entire testing data with and without safety margin. These numbers reflect how often clients had to wait for sleeping PMs to be switched ON before their requests got allocated. Since the Min Predictor had the least error among the approaches we compared against in Fig. 2.7, we also report in Fig. 2.10 the total number of unpredicted PMs when the Min Predictor is used in our framework to make predictions. Observe that the adaptive Wiener filter without safety margin had a lower number of unpredicted PMs compared to the Min Predictor. The results also show that by adding a safety margin to the predictions of the Wiener filter, the number of unpredicted PMs that were switched ON was reduced further by 87% compared to the case without safety margin.

### 2.7 Power Management

As mentioned previously in Section 2.3, the Power Management module keeps track of all the PMs in the cloud cluster and stores their capacities, their current utilization and their state (ON or sleep) in a table, referred to as the ResourceTable. Let $\hat{d}(i)$ be the predicted
Figure 2.10: Total number of unpredicted waken PMs during the entire testing period.

number of requests of the $i^{th}$ category in the coming prediction window (taken from the output of the workload predictor); and $c_i$ be the center of the $i^{th}$ category (taken from the data clustering stage). The Power Management module relies on ResourceTable along with the centers and the predicted number of requests for each category to decide on the number of needed PMs in the coming prediction window. Algorithm 1 shows the heuristic approach used by the Power Management module to determine the number of needed PMs. The steps of the algorithm are as follows:

In line 1, a copy of ResourceTable called PMTable is made to be used as a draft table to determine the number of needed PMs. This draft table is made so that the state and the utilization fields of the original ResourceTable are not changed while trying to fit the predicted VM requests in the PMs in order to estimate the number of needed PMs. In line 2, PMTable is sorted based on the three criteria introduced in Section 2.3.3. In line 3, VMList is a list constructed to contain all the predicted requests in the coming prediction window. In fact, VMList contains $\bar{d}(1)$ VM request with the CPU and memory resources specified by $c_1$, $\bar{d}(2)$ VM request with the resources specified by $c_2$ and so on for all the categories. In line 4, the requests in VMList are sorted from the largest to the smallest request. Since each request has two resources (CPU and memory), the product
of the requested CPU and memory is used as a metric when sorting the requests. The intuition behind this sorting criteria is to consider VMs with larger CPU and memory demands for placement first as they are harder to fit since they require larger free space. It is worth mentioning that the product of the requested CPU and memory resources is just used as a sorting metric and the amount of requested CPU and memory resources are actually allocated to each VM request when it is placed on a PM.

The algorithm iterates on the sorted VMs and starts by picking the first VM request in the ordered $\text{VMList}$. Next, it iterates over the sorted PMs trying to fit the picked VM within the first PM in the ordered $\text{PMList}$. The algorithm keeps iterating over the PMs in $\text{PMList}$ until the VM request fits one. The PM that fits the picked VM can be either asleep or ON. If it is asleep, then the PM is added to the $\text{ONList}$ (line 9) and the PM’s state is changed to ON (line 10). $\text{ONList}$ is basically a list that stores the PMs that need to be turned ON to cover the future workload. If the picked PM is not asleep, these two steps are skipped and the algorithm goes directly to line 12. Since the PM will host the picked VM request, the utilization of the PM is updated by calculating the new utilization of the PM after placing the picked VM request (line 12). In line 13, $\text{PMList}$ is sorted based on the three criteria since at least the utilization (or both the utilization and state) of one of the PMs is changed. The algorithm picks next the following VM request in $\text{VMList}$ and tries to fit it similarly in a PM, and repeats the same procedure until all VMs are placed in a PM.

After assigning all of the predicted VMs to a PM, the algorithm checks whether there are PMs that are ON but have zero utilization. If that is the case, then these PMs are added to $\text{SleepList}$ as they are redundant and need to be switched to the sleep mode (line 20). This case might happen if the cloud clients terminated many VM requests within the last minute and the corresponding PMs that host these VMs became idle.
If the predicted workload in the future prediction window can be hosted by a subset of these idle PMs, then the extra PMs are switched to the sleep state to save energy.

The algorithm ends up with two lists: `ONList` and `SleepList` where one of the two lists is empty. It is easy to see that one of the two lists will be empty, as in our PM sorting criteria, machines that are ON come first and thus will be used to host the predicted VMs. More PMs will be turned ON only when all already ON PMs are used; in this case, `SleepList` will be empty as there will be no ON PMs with zero utilization. Finally, the Power Management module turns all the PMs in `ONList` ON or switches all the PMs in `SleepList` to the sleep state and updates the new states in the original `ResourceTable` respectively. As a result, we end up keeping ON only the needed machines to cover the predicted workload, and putting to sleep the rest of machines.

2.8 Framework Evaluation

In this section, we evaluate the performance of our integrated energy-aware cloud resource provisioning framework (with adaptive prediction and safety margin) and compare it against the following schemes\textsuperscript{2}:

- **No Power Management**: represents the case where all PMs of the cluster are left ON all the time.

- **PM Prediction Power Management**: this scheme follows the prediction approach that was proposed in [49, 50] which observes the number of PMs that were needed in each minute during the last $T_{obs}$ minutes, predicts the average of those observations to be the number of PMs needed in the coming minute, and scales the

\textsuperscript{2}The number of ON PMs used by the Google cluster is kept private and thus we could not compare with Google’s power management scheme.
Algorithm 1: Modified BFD – estimating the number of needed PMs during the next prediction window

Input:
ResourceTable: a table whose entries each corresponds to a PM and contains its power state, its CPU and memory capacity and its CPU and memory utilization.
c_1, c_2, ..., c_k: centers of the k categories. Each center is a point in \( \mathbb{R}^2 \) where the two dimensions are the CPU and memory resources.
\( \bar{d}(1), \bar{d}(2), ..., \bar{d}(k) \): predicted numbers of requests for the k categories in the coming prediction window.

Output:
SleepList: List of PMs to be switched to sleep mode.
OnList: List of PMs to be turned on to cover the predicted workload.

1: PMTable ← ResourceTable
2: PMTable.sortPMs()
3: VMList ← ConstructList(\( \bar{d}(1), \bar{d}(2), ..., \bar{d}(k), c_1, c_2, ..., c_k \))
4: VMList.sortVMs()
5: for each VM in VMList do
6:   for each PM in PMTable do
7:     if VM fits in PM then
8:       if PM was asleep then
9:         OnList.add(PM)
10:        PMTable.SetState(PM,"ON")
11:     end if
12:    PMTable.UpdateUtilization(PM,VM)
13:   end if
14:   break the for loop and try to pack the next VM
15: end for
16: end for
17: end for
18: for each PM in PMList do
19:   if PM.state="ON" AND PM.utilization=0 then
20:     SleepList.add(PM)
21:   end if
22: end for
number of ON PMs up or down based on this prediction. A fixed number $N_{buffer}$ of extra PMs is left ON as a buffer on top of those predictions to reduce the cases where clients have to wait for a sleeping PM to be switched ON before their requests get allocated. Tuning the parameters of this scheme ($T_{obs}$ and $N_{buffer}$) is done by selecting the values that achieved the best prediction accuracy on the training data.

- **Optimal Power Management:** represents the case where the predictor knows the exact number of future VM requests, as well as the exact amount of CPU and memory resources associated with these requests. This represents the best-case scenario for resource management and serves here as an upper bound.

2.8.1 Resource Utilization

We start first by analyzing how efficient the ON PMs are utilized when the different schemes are used to manage the Google cluster. Figs. 2.11 and 2.12 show snapshots that plot the average CPU and memory utilization of the ON PMs over time when evaluating the different schemes on the testing period. For completeness, we also show in Table 2.4 the ON PMs average CPU and memory utilizations averaged over the entire duration of the testing data. Observe that both CPU and memory utilizations are very low when no power management is applied, since many PMs are kept ON without being utilized. The PM Prediction Power Management scheme improves the utilization of the CPU and memory resources compared to the previous scheme, as it observes how many PMs were recently used and predicts broadly how many ON PMs will be needed in future. However, this scheme still wastes resources and is far from the optimal utilization levels.
Our framework, on the other hand, improves further the average utilization of both CPU and memory, as it estimates accurately how many ON PMs will be needed by predicting the future requests along with their requested resource demands. In fact, the utilization achieved under our framework is very close to that of the Optimal Power Management scheme. The utilization gap between our framework and the optimal case is mainly due to prediction errors and to the redundant PMs that are left ON as a safety margin. It is worth mentioning that the optimal power management does not achieve 100% utilization due to the bin packing nature, where some PMs may still have some space left (not fully utilized).
Figure 2.13: A comparison of the consumed energy.

Figure 2.14: Total consumed energy during the entire testing period.
Table 2.4: Avg. CPU and Memory Utilization over all testing data.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AvgCPU Util</td>
<td>38.15 %</td>
<td>59.06 %</td>
<td>83.94 %</td>
<td>90.42 %</td>
</tr>
<tr>
<td>AvgMem Util</td>
<td>32.69 %</td>
<td>50.37 %</td>
<td>65.34 %</td>
<td>69.92 %</td>
</tr>
</tbody>
</table>

2.8.2 Energy Savings

We now assess the total energy consumed by the Google cluster when the traces reported during the testing period are handled by the different resource management schemes. The cluster’s total costs include the energy consumed by both ON and sleeping PMs, as well as the transition energy associated with switching a PM from ON to sleep and vice versa. We rely on the power numbers reported in [48] and summarized in Table 2.3 to calculate those total costs where the power consumed by a sleeping PM is $P_{\text{sleep}}$, whereas the power consumed by an ON PM increases linearly from $P_{\text{idle}}$ to $P_{\text{peak}}$ as its CPU utilization increases from 0 to 100%.

Fig. 2.13 shows a snapshot of the energy consumed by the cluster when it is handling the requests reported in the testing traces using the different resource management schemes. Observe that by leaving all the PMs ON, the No Power management scheme consumes significant amount of energy over time. By predicting on average how many PMs will be needed using the PM Prediction scheme, the cluster reduces its costs. Whereas by using our proposed framework, the consumed energy is lower than the former two cases as we are decomposing the workload into multiple categories, and predicting the workload for each category which helps in keeping the right amount of needed servers ON. Furthermore, unlike the PM Prediction scheme which leaves a fixed number of ON PMs as a buffer all the time, our framework uses an adaptive safety mar-
gin that is proportional to the workload predictions, which avoids keeping too many ON PMs when they will not actually be needed. We also show in Fig. 2.13 the energy costs of the Optimal Power Management scheme. Results show that the energy costs of our framework are slightly larger than the optimal case. This difference is mainly due to the prediction errors and to the safety margin overhead.

For completeness, Fig. 2.14 shows the total energy consumed by the Google cluster during the entire testing period for the different schemes normalized with respect to the total energy cost of the No Power Management scheme. The results show how close the energy consumed by our framework is from the optimal case and highlights that a significant portion of the cluster’s consumed energy can be saved by estimating accurately the future workload. It is worth mentioning that the energy costs associated with the optimal scheme are inevitable and represent the energy consumed by the hosted workload.

We discussed so far the amount of energy that our framework saves. Now in order to have a sense/estimate of the actual savings in terms of money, observe from Fig. 2.13 that about 50 Mega Joules are saved by our framework every minute when compared to the PM Prediction scheme. This translates into a total savings of 2160 Giga Joules per month, or equivalently a total of 600 Megawatt-hour per month. Per year, this translates into a saving of 7200 Megawatt-hour. For example, in California in 2013, the commercial cost of one kilowatt-hour is about 17 Cents [51]. Therefore, we can say that our techniques can save the studied Google cluster roughly about $1.2 million per year.
2.8.3 Workload Underestimation Evaluation

We assess now how often clients had to wait for a sleeping PM to be switched ON before their requests got allocated. In order to do that, we show in Table 2.5 the total number of underestimated PMs that were switched from sleep to the ON state to accommodate the received workload when each of our framework and the PM Prediction Power Management scheme were evaluated on the entire testing data. Clearly, the number of clients that had to wait before their resources were allocated is proportional to the number of underestimated PMs that had to be awaken from sleep to accommodate the received requests. Observe that our framework had around 65% less underestimation incidents compared to the PM Power Management scheme. This proves the efficiency of our prediction and safety margin techniques compared to the PM Prediction scheme which calculates broadly how many ON PMs will be needed and keeps a fixed number of ON PMs as a buffer to avoid those underestimation cases.

<table>
<thead>
<tr>
<th></th>
<th>PM Pred. Mngmt</th>
<th>Our Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of PMs</td>
<td>1270</td>
<td>456</td>
</tr>
</tbody>
</table>

2.8.4 Execution Time Overhead Evaluation

Our last experiment analyzes the execution time overhead of the offline as well as the online stages of our framework. All the measurements reported in this section are based on running our framework on a platform that has an Intel Core 2 Quad Q9400 2.66 GHz processor, and an 8 GB RAM.

As for the offline stages, they are executed only once on large training traces in order
to capture the general workload characteristics. Recall that our framework has two offline stages: the Clustering stage (used to decide how many categories the requests will be divided to), and the the Predictors Training stage (used to tune initially the parameters for each category’s predictor). Table 2.6 shows the total time spent to execute each one of those offline stages on the Google training data.

Online stages on the other hand are performed periodically every minute. We measure how much time each online stage takes each time it is called when running our framework on the Google testing data. Since the time spent by each online stage may slightly vary depending on the workload, we report in Table 2.6 the mean and standard deviation for how much time each online stage takes when it is called by our framework.

The reported measurements in Table 2.6 clearly show that our framework not only achieves great savings but also has a light offline and online execution overhead, which is something important for efficient resource management.

<table>
<thead>
<tr>
<th>Table 2.6: Execution time analysis of our framework’s stages.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline Stage</td>
</tr>
<tr>
<td>Clustering</td>
</tr>
<tr>
<td>Predictors Training</td>
</tr>
<tr>
<td>Online Stage</td>
</tr>
<tr>
<td>Trace Decomposition</td>
</tr>
<tr>
<td>Workload Prediction</td>
</tr>
<tr>
<td>Power Management</td>
</tr>
</tbody>
</table>

2.9 CONCLUSION AND FUTURE WORK

We propose in this chapter an integrated energy-aware resource provisioning framework that predicts the number of VM requests and the amount of resources associated with these requests that the cloud center will receive in the future. These predictions are
used to keep the right amount of needed PMs ON. The methods to estimate the parameters involved in our framework are presented including setting the number of clustered categories, the length of the prediction window, the optimal weights of the stochastic predictor, and the length of the observation window. The effectiveness of our framework is evaluated using real traces from Google cloud cluster. We show that our framework outperforms existing prediction techniques and achieves significant energy savings and high utilization that are very close to the optimal case. For future work, we plan to investigate whether VM requests follow certain regular daily trends and rely on that to further improve the workload prediction module. Finally, the lack of publicly available fine-grained traces, similar to Google traces, has prevented us from testing our framework on other cloud traces. We hope to have the chance to do that in future.
Chapter 3: Energy Efficient Resource Allocation and Scheduling

Framework

We exploit in this chapter cloud task elasticity and price heterogeneity and propose an online resource management framework that maximizes cloud profits while minimizing energy expenses. This is done by reducing the duration during which servers need to be left ON and maximizing the monetary revenues when the charging cost for some of the elastic tasks depends on how fast these tasks complete, while meeting all resource requirements. Comparative studies conducted using Google data traces show the effectiveness of our proposed framework in terms of improving resource utilization, reducing energy expenses, and increasing cloud profits.

3.1 RELATED WORK

Maximizing profits while meeting clients’ requirements is what cloud providers aim for. Large electricity bills are being paid by cloud providers [52], merely due to a poor management of cloud resources [53, 54], resulting in having servers stay unnecessarily ON for long periods while being under-utilized [14]. Therefore, to minimize the cloud’s consumed energy, one needs to reduce the number of ON servers, increase the servers utilization and reduce the duration for which servers are left ON while meeting clients’ demands.

A cloud center is made up of a huge number of servers called physical machines (PMs) that are grouped into multiple management units called clusters. Cloud clients
may, at any time, submit a VM request to the cloud, specifying the amount of computing resources they need to perform a certain task.

Prior cloud resource management schemes focused on handling task requests of the form \((w_{req i}^{i}, t_{req i}^{i})\) where \(w_{req i}^{i}\) is task \(i\)'s requested amount of CPU resources and \(t_{req i}^{i}\) is the duration for which these resources are needed. Such requests are referred to by *inelastic* task requests as they require a fixed amount of CPU resource during their whole execution time and as increasing their allocated resources at any time would not decrease their execution times. We define the requested computing volume \(v_{req i}^{i}\) for an inelastic task \(i\) as \(v_{req i}^{i} = w_{req i}^{i} \times t_{req i}^{i}\).

A cloud task request could also be *elastic* in that the amount of its allocated resources can be increased (or decreased) during its execution time, and doing so results in reducing (or prolonging) its execution time. An example of such tasks would be any task with thread parallelism where the number of allocated threads (CPU resources) determines how fast the task completes. An elastic task request \(i\) is defined by \((v_{req i}^{i}, w_{max i}^{i}, t_{max i}^{i})\) where \(v_{req i}^{i}\) is the requested computing volume, \(w_{max i}^{i}\) is the maximum amount of CPU resources that can be allocated to the task, depending primarily on its maximum amount of parallelism, and \(t_{max i}^{i}\) is a duration period by which the task should complete. Note that the least duration needed to complete such a request is \(t_{min i}^{i} = v_{req i}^{i} / w_{max i}^{i}\), which corresponds to allocating the maximal amount of resources \(w_{max i}^{i}\) all the time. Thus, \(t_{max i}^{i}\) must be strictly larger than \(t_{min i}^{i}\) for such a request to be feasible and elastic.

Two service models arise for handling elastic tasks:

i) **Best-Effort Model**: clients are only interested in getting their task completed within a specified period \(t_{max i}^{i}\), and are charged a fixed cost based on their task’s volume.

ii) **Delay-Sensitive Model**: not only does the task’s deadline have to be met, but also clients are willing to pay more to get their task completed earlier.
The elasticity among the tasks brings further management capabilities. Exploiting these capabilities requires developing efficient scheduling techniques that decide: \( i \) where to place the heterogeneous tasks, and \( ii \) how much resources to allocate for each elastic task. The latter decision has a significant impact on the cloud’s monetary profits. This point can be illustrated by considering a simplified example where the cloud receives two elastic task requests, \( T_1 \) and \( T_2 \), each requesting a volume of \( v^{req} = 1.5 \), each can be allocated at most \( w^{max} = 75\% \) of the PM’s capacity, and each must complete within \( t^{max} = 6 \) hours. Assume that both tasks were assigned to the same PM which has a unit capacity. There are numerous feasible ways to allocate resources for \( T_1 \) and \( T_2 \), and we are seeking those that maximize the cloud profits. From an expense point of view, the allocation in Fig. 3.1(a) meets the tasks requirements with the least cost among all possible allocations as it keeps the PM ON for the least time (3 hours only), after which the PM is turned to sleep to save energy. Of course finding the allocation with the least expenses is harder than this simplified example in real cases when many tasks (elastic and inelastic) with different computing demands are cohosted on the same PM. Things become further challenging when some of the hosted tasks are delay-sensitive as the resource allocation not only impacts the energy expenses but also determines the collected revenues. By returning to our example and considering only \( T_1 \) to be a delay-
sensitive task, observe that although the allocation in Fig.3.1(a) have lower costs than that in Fig.3.1(b), the latter allocation has higher revenues than the former case as $T_1$ completes as fast as possible charging the client the highest cost. Thus the allocation in Fig.3.1(b) will be more profitable than that in Fig.3.1(a) only if the extra revenue collected for completing $T_1$ earlier outweighs the energy cost resulting from keeping the machine on for the entire 6 hours (as opposed to 3 hours only). This makes managing cloud resources in a profitable way a challenging task.

In this chapter, we exploit the elasticity and the varying charging costs among the submitted tasks and propose an online resource allocation framework that aims at maximizing the cloud’s profits. To the best of our knowledge, this is the first work that exploits both of those aspects. Our main contributions are in proposing a framework that:

1. places the submitted elastic and inelastic tasks in a way that avoids turning new PMs ON while also reducing the PMs’ uptimes (i.e., the duration for which PMs need to be kept ON to serve the hosted tasks).

2. decides what amount of resources should be assigned initially to the elastic tasks while guaranteeing that their deadlines will be met.

3. tunes the amount of allocated resources for the elastic tasks over time by solving a convex optimization problem whose objective is to maximize the cloud’s profits.

The remainder is organized as follows. Section 3.1 reviews prior work. Section 3.2 presents our proposed framework. Section 3.3 proposes charging models for the elastic services. Section 3.4 describes the optimization problem solved by our framework to tune the allocations over time. Section 3.5 evaluates our framework on real Google traces.
Finally, Section 3.6 concludes the chapter and Section 3.7 provides directions for future work.

3.2 THE PROPOSED FRAMEWORK

As illustrated in Fig. 3.2, our proposed resource allocation framework has a two-level control structure as it is made up of a front end VM Placement module connected to all the PMs in the cloud cluster and an autonomous Resource Management module dedicated to each PM in the cluster. We explain next each one of these control modules:

3.2.1 VM Placement Module

Upon receiving a task request (elastic or inelastic), this module creates a VM for the submitted task and decides what PM in the cluster the created VM should be assigned to. These placement decisions are made depending on the current states of the PMs in the cluster and on two task-related quantities: the amount of CPU resources assigned
initially to the created VM, and the time after which the VM is expected to be released.
In the case of an inelastic task, these two quantities are directly specified by the client to be \( w_i^{req} \) and \( t_i^{req} \) respectively. On the other hand, there is flexibility in these quantities if the task is elastic as they are only bounded above by \( w_i^{max} \) and \( t_i^{max} \) respectively. Thus, the module needs to decide the quantity of resources that should be assigned initially to the elastic task in order to make efficient PM placement. Our module allocates \( w_i^{min} = \frac{v_i^{req}}{t_i^{max}} \) amount of resources to the elastic task initially. This represents the least amount of resources needed so that the task is accomplished exactly in the client’s maximum tolerable period \( t_i^{max} \). The intuition behind this choice is the following. If we allocate less than \( w_i^{min} \) initially, then at some point in time we need to increase these resources so that the task terminates within \( t_i^{max} \). However, the PM’s capacity constraint and the constraints from the remaining hosted VMs may prevent us from doing that and thus we may risk to miss the task’s deadline. Thus at least \( w_i^{min} \) amount of resources should be allocated initially to avoid that. On the other hand, if an amount of resources larger than \( w_i^{min} \) is assigned to the VM, then there may be no ON PM with enough slack to fit that VM and we will be forced to switch a new PM from sleep to fit this VM. This switch has a high energy overhead [55, 56] and also increases the number of ON PMs which leads to high energy consumption. This costly switch will be unnecessary if an ON PM with a slack of only \( w_i^{min} \) is already available in the cluster. Thus \( w_i^{min} \) is initially allocated to the elastic task in order to guarantee meeting its deadline while also saving energy. It is worth mentioning that these are only initial assignments for the elastic tasks and will be later tuned in order to maximize the cloud profits as will be explained later. Having determined the amount of resources to be allocated initially to the submitted task’s VM and when the VM is expected to be released, we now explain the PM preference criteria that is adapted for efficient placement selection. Based on our
preference criteria, the PMs in the cluster are divided into the following disjoint groups:

1. **Group 1:** contains all the PMs that are currently ON and that have an uptime larger than the VM’s release time.

2. **Group 2:** contains the remaining ON PMs (the ones with an uptime smaller than the VM’s release time).

3. **Group 3:** contains all the PMs in the sleep state.

The module tries first to place the new task’s VM in one of the PMs of Group 1. These PMs are mostly favored as their uptime will not be increased after placing the new VM. If multiple PMs of Group 1 can fit the VM, then the one with the least CPU slack is chosen. The intuition behind this preference is to leave larger slacks on the remaining ON PMs so that VMs with larger CPU demands can be supported by these PMs in the future without needing to wake new PMs from sleep. If none of the PMs in Group 1 have enough space to fit the VM, then the PMs of Group 2 are considered. If multiple PMs from Group 2 can fit the VM, then the one whose uptime will be extended the least after placement will be chosen. This is to reduce the extra time for which the PM will be kept ON. Finally, if no fit is found in Group 2, then Group 3 is considered. Thus our preference criteria switches a new PM ON to accommodate the VM only if we have no other choice. The PM in Group 3 with the largest capacity is chosen in that case so that requests with larger demands can be supported by this PM in the future.

### 3.2.2 Resource Management Module

The main role of this module is to make resource and power management decisions for the controlled PM. A PM $P_j$’s module performs the following procedures:
1. Resource Allocation: A flag is dedicated to indicate whether or not the allocated resources for the tasks hosted on the controlled PM need to be retuned. This flag is set true whenever one of the following events occurs: a) a new delay-sensitive elastic task gets placed on the PM, or b) the PM’s uptime gets increased due to assigning a new best-effort elastic task, or c) one of the already hosted tasks (elastic/inelastic) gets released from the PM. The resource management module checks this flag periodically every $T_p$ period. If the flag is true, then the amount of resources $w_i$ to be allocated to each task $i$ hosted on $P_j$ is tuned. This is done by solving an optimization problem whose objective is to maximize the PM’s profit, which is calculated to be the difference between the revenues collected from serving the hosted tasks and the PM’s energy expenses spent to accomplish those tasks, and where constraints are included to guarantee meeting all tasks’ requirements and deadlines. Details on the resource allocation optimization problem are provided in Section 3.4. The flag is reset to false after updating these allocations. Ideally, to maximize the profits, new resource tuning needs to be calculated anytime a new delay-sensitive elastic task gets placed on the PM (as it might be more profitable to finish this task earlier), or any time the PM’s uptime gets extended due to assigning a new best-effort elastic task (as it might be possible to increase the allocated resources for this task so that the PM’s uptime gets decreased in order to save energy), or anytime a task gets released (as an extra free resource slack becomes available to use for the remaining hosted tasks). However, tuning the allocated resources for the hosted tasks each time one of those events occur has a high computation overhead. The flag is thus introduced for practical uses in order to limit the number of times the optimization problem is solved and the resources are retuned per PM to be once at most every $T_p$ period. The choice of $T_p$
is left to the cloud provider depending on how much computation overhead it can afford. The smaller the selected value for this parameter, the more responsiveness the framework becomes to tune allocations that maximize its profits, but also the higher the computation overhead. In our implementation, $T_p$ is set to 5 minutes as evaluations revealed that high revenues and great energy savings can be achieved for such choice while keeping the calculation overhead low.

2. Remaining Time/Volume Tracking: for each elastic task $i$ hosted on $P_j$, the module tracks $t_{rem}^i$, the amount of remaining time before which the task should be accomplished. The remaining time is initially set to $t_{i, max}^i$ when the task is first scheduled on the PM and is decremented as time goes by. For each elastic or inelastic task $i$ hosted on $P_j$, the module also tracks the amount of remaining computing volume still needed to accomplish each one of these tasks. The remaining computing volume for task $i$, referred to by $v_{i, rem}^i$, is initially set to be equal to the task’s requested computing volume $v_{i, req}^i$ when the task is first scheduled on $P_j$. Let $w_i$ be the amount of resources allocated to task $i$ for period $T$, then the remaining volume will be updated after the $T$ period is over as follows: $v_{i, rem}^i \leftarrow v_{i, rem}^i - (w_i \times T)$.

3. VM Termination: a task completes when it is allocated all of its requested computing volume. Thus the module releases the allocated resources for VM $i$ hosted on the PM once its remaining computing volume $v_{i, rem}^i$ reaches zero. If no other VMs are still hosted on the PM after this release, then the PM is put to sleep to save energy.
3.3 Charging Models

We first discuss how inelastic tasks are being charged in current cloud centers, and then propose a charging model for both the best-effort and the delay-sensitive elastic tasks.

In current cloud providers (e.g. Amazon, Microsoft), the charging cost for an inelastic task $i$ is dependent on the task’s volume $v_{i}^{req}$ which captures both the amount of requested resources $w_{i}^{req}$ and the duration $t_{i}^{req}$ for which these resources are reserved.

More specifically, letting $\varphi$ denote the price per a unit of volume for the inelastic service, the charging cost $r_{i}$ for the inelastic task $i$ can be expressed as

$$r_{i} = \varphi v_{i}^{req}$$

We propose a similar model for the best-effort elastic tasks with the only exception that the cloud provider charges the client’s task with a reduced price compared to the inelastic case. This is basically to provide an incentive for clients to request an inelastic service as clients in this case provide a flexibility in terms of the allocated resources and in terms of when their tasks can be accomplished. This flexibility provides further management capabilities that the cloud provider exploits to reduce its expenses and increase significantly its profits (as will be seen in Section 3.5), so that the reduced price for this service ends up to be a win-win situation for both the clients and the cloud provider. More specifically, the charging cost $r_{b}^{min}$ for a best-effort elastic task $b$ with a computing volume $v_{b}^{req}$ is calculated as follows:

$$r_{b}^{min} = \phi v_{b}^{req}$$

where $\phi$ is the reduced cost per one unit of volume which is specified by the cloud provider.
and where $\phi < \varphi$ holds. Observe that the cost of this type of service is only dependent on the task’s volume where the cloud provider guarantees providing the requested volume to complete the task within the maximal tolerable duration $t_b^{\max}$. It is worth mentioning that in our model the reduced price $\phi$ was considered to be fixed for all the elastic requests. Another option could be to have a reduced price $\phi$ that is different from an elastic task to another depending on how flexible each elastic task is. The higher the elastic task’s flexibility, the lower the charged cost and vice versa.

As for the delay-sensitive elastic tasks, the charging cost in that case is not only dependent on the task’s volume but also is affected by how fast the task completes. We propose a linear charging model for this service, where the charging cost $r_d$ of the delay-sensitive elastic task $d$ is expressed as a function of the task’s duration $t_d$ as follows:

$$r_d(t_d) = r_d^{\min} + \delta(t_d^{\max} - t_d) \quad (3.3)$$

Where: $r_d^{\min}$ captures the volume cost and is calculated using equation (3.2), and $\delta$ is the price that the client pays for getting his task completed one unit of time earlier and is specified by the cloud provider. Recall that the duration by which the task completes $t_d$ is bounded up by $t_d^{\max}$ which is the maximal tolerable duration that is specified by the client’s request. There is also an implicit lower bound on $t_d$ as the task can be allocated at most $w_d^{\max}$ of CPU resources at any time, and thus the least duration needed to complete the task is $t_d^{\min} = v_d^{\text{req}} / w_d^{\max}$ which corresponds to allocating the maximal resources $w_d^{\max}$ all the time. Observe that based on equation (3.3) as the task’s duration increases from $t_d^{\min}$ to the maximum tolerable duration $t_d^{\max}$, the charging cost for the delay sensitive task decreases linearly from $r_d^{\max} = r_d^{\min} + \delta(t_d^{\max} - t_d^{\min})$ to $r_d^{\min}$.

To further illustrate our proposed charging models, Fig. 3.3 plots how an elastic
task $i$ would be charged as the duration $t_i$ needed to complete the elastic task increases from $t_{i\text{min}}$ to $t_{i\text{max}}$ for both the best-effort and the delay-sensitive services. Observe that if task $i$ requested a best-effort service, then its charging cost would be the same (equals to $r_{i\text{min}}$) regardless of when the task completes, as long as the task completes within the maximal duration $t_{i\text{max}}$ specified by the client. Whereas if task $i$ requested a delay-sensitive service, then its charging cost would be dependent on how fast the task completes where the cost decreases linearly from $r_{i\text{max}}$ to $r_{i\text{min}}$ as the duration needed to complete the task $t_i$ increases from $t_{i\text{min}}$ to $t_{i\text{max}}$. It is worth noting that the slope of the charging cost for the delay-sensitive services is controlled by $\delta$.

### 3.4 PM Resource Allocation

Having explained the charging models for the elastic tasks, we now elaborate on how our proposed resource management module allocates resources for the tasks that are hosted on a PM $P_j$ which has a certain CPU capacity $C_j$. Let $\mathbb{E}_j$ and $\mathbb{I}_j$ be respectively the sets of all elastic and inelastic tasks currently hosted on $P_j$. Recall that each task in $\mathbb{E}_j$ is either a best-effort or a delay-sensitive elastic task. Let $\mathbb{B}_j$ and $\mathbb{D}_j$ be respectively the sets of the best-effort and the delay-sensitive elastic tasks that are hosted on $P_j$ where
\[ E_j = B_j \cup D_j. \] We explain in this section how to tune \( w_i \in \mathbb{R}^{++} \), the amount of CPU resources to be allocated to each task \( i \) hosted on PM \( P_j \). The allocated resources \( w_i \) allows, in turn, to determine \( t_i \in \mathbb{R}^{++} \), the time needed to accomplish the \( i \)th task. Our framework allocates resources to the tasks assigned to \( P_j \) by solving the following optimization problem.

### 3.4.1 Formulated Optimization Problem

**Objective Function:** The objective of our resource allocation strategy is to maximize the PM’s profits which can be calculated as the difference between the PM’s revenues \( R_j \) and the PM’s energy expenses \( X_j \), i.e., we seek to:

\[
\text{Maximize } R_j - X_j
\]

We elaborate now how the PM’s revenues and expenses are calculated based on the allocated resources:

1. **PM Revenues:** can be calculated by aggregating the revenue collected from each task hosted on \( P_j \), i.e.:

\[
R_j = \sum_{i \in I_j} r_i + \sum_{b \in B_j} r_{b_{\text{min}}} + \sum_{d \in D_j} r_{d(t_d)}
\]  
   (3.4)

Where the first, second and third summations aggregate respectively the revenue collected from the inelastic tasks, the best-effort elastic tasks and the delay-sensitive elastic tasks that are hosted on the PM.

2. **PM Expenses:** Since \( P_j \) needs to be kept ON until its last hosted task is accom-
plished, then the PM’s energy expenses can be calculated as:

$$X_j = \kappa \times \max\{t_i : i \in E_j \cup I_j\}$$

where: $\kappa$ is the cost to keep the PM ON for a single unit of time and depends primarily on the PM’s power specs and the electricity price. More specifically, $\kappa = \xi \times P_{\text{active}}$ where: $\xi$ is the electricity price (given in dollars/unit of energy) and $P_{\text{active}}$ is the power consumed to keep the PM ON.

**Constraints:** The optimization problem is solved subject to the following constraints. One,

$$\sum_{i \in E_j \cup I_j} w_i \leq C_j \quad (C.1)$$

which states that the aggregate allocated resources for all the tasks hosted on $P_j$ must not exceed the PM’s capacity.

Two,

$$w_i = w_i^{\text{req}} \quad \forall i \in I_j \quad (C.2)$$

which states that any inelastic task request must be assigned the exact amount of requested CPU resources. Three,

$$w_i \leq w_i^{\text{max}} \quad \forall i \in E_j \quad (C.3)$$

which states that the allocated resources for any elastic task must not exceed the maximum amount of resources that the task supports. Four,

$$t_i \leq t_i^{\text{rem}} \quad \forall i \in E_j \quad (C.4)$$
which states that the accomplishment time of each elastic task must be within $t_i^{rem}$, the
remaining duration before which the task should be accomplished.

Five,

$$t_i \times w_i = v_i^{rem} \quad \forall i \in \mathbb{E}_j \cup \mathbb{I}_j$$  \hspace{1cm} (C.5)

which states that the resulting allocations must provide the remaining computing vol-
umes $v_i^{req}$ required by the hosted tasks.

### 3.4.2 Equivalent Optimization Problem

We introduced in the previous subsection the objective and the constraints of the for-
mulated resource allocation optimization problem where the decisions variables for each
task $i$ are $w_i \in \mathbb{R}^{++}$ and $t_i \in \mathbb{R}^{++}$. We show now how to perform simple transformation
on the above introduced problem in order to transform it into an equivalent problem
of a certain type whose optimal solution can be obtained easily. The transformation
consists basically of performing a variable renaming by letting $t_i = 1/f_i$ where the de-
cision variables for each task $i$ become now $w_i \in \mathbb{R}^{++}$ and $f_i \in \mathbb{R}^{++}$. This changes the
optimization problem as follows.

**Objective Function After Transformation:** Both the revenues and the expenses
of the objective are affected by this variable renaming as follows:

1. *PM Revenues:* only the revenue collected from the delay-sensitive elastic tasks is
   affected by this renaming where (3.4) becomes:

   $$\mathcal{R}_j = \sum_{i \in \mathbb{I}_j} r_i + \sum_{b \in \mathbb{B}_j} r_b^{min} + \sum_{d \in \mathbb{D}_j} r_d(1/f_d)$$
After plugging (3.1), (3.2) and (3.3) in the previous expression, we end up with:

\[ R_j = \sum_{i \in I_j} \phi v_i^{req} + \sum_{b \in B_j} \phi v_b^{req} + \sum_{d \in D_j} \phi v_d^{req} + \delta(t_{d_{\text{max}}} - 1/f_d) \]

By performing simple algebraic manipulation, the PM revenue can be expressed as:

\[ R_j = R_j^{\text{const}} - \delta \sum_{d \in D_j} 1/f_d \quad (3.5) \]

Where:

\[ R_j^{\text{const}} = \sum_{i \in I_j} \phi v_i^{req} + \sum_{b \in B_j} \phi v_b^{req} + \sum_{d \in D_j} \phi v_d^{req} + \delta t_{d_{\text{max}}} \]

is a constant as it is not affected by any of the optimization problem decision variables. In other words, any feasible solution for the formulated optimization problem allocates for the inelastic tasks the requested CPU resources for the specified period of time and also guarantees meeting the deadline of all the elastic tasks, thus the revenues collected from the inelastic and from the best-effort elastic tasks are the same for any feasible solution. From a revenue point of view, what differentiate a feasible solution from another is only the revenue collected from the delay-sensitive elastic tasks, which is captured by the second term of (3.5). The PM revenues \( R_j \) calculated based on (3.5) is a concave function. This can be proved based on the following facts [57]. First, for any given task \( d \), the function \( 1/f_d \) is convex on \( \mathbb{R}^{++} \). Second, the positive summation of a set of convex functions is also a convex function (thus \( \sum_{d \in D_j} 1/f_d \) is convex). Third, multiplying a convex function by a negative scalar \((-\delta)\) reverses the curvature from convex to concave. Finally, adding a constant \((R_j^{\text{const}})\) to a concave function preserves the concave curvature.
2. **PM Expenses**: becomes after renaming:

\[ \mathcal{X}_j = \kappa \times \max\{1/f_i : i \in E_j \cup I_j\} \]

The PM expenses \( \mathcal{X}_j \) after variable renaming is a convex function based on the following facts [57]. First, for any given \( i \), the function \( 1/f_i \) is convex on \( \mathbb{R}^{++} \). Second, the maximum of a set of convex functions is a convex function (thus \( \max\{1/f_i : i \in E_j \cup I_j\} \) is convex). Finally, multiplying a convex function by a positive scalar \( \kappa \) maintains the convex curvature.

Putting it all together, the objective function \( R_j - \mathcal{X}_j \) that we seek to maximize after transformation is the difference between the concave function \( R_j \) and the convex function \( \mathcal{X}_j \). Since \( \mathcal{X}_j \) is convex, then \( -\mathcal{X}_j \) is concave. Now since the positive summation of two concave functions is concave, then the objective function \( R_j - \mathcal{X}_j \) is a concave function that we seek to maximize. Finally, the problem of maximizing the concave function \( R_j - \mathcal{X}_j \) can be reformulated equivalently as a problem of minimizing \( -(R_j - \mathcal{X}_j) \), which is convex.

**Constraints After Transformation**: Constraints (C.1), (C.2), and (C.3) remain the same after the variable renaming as they are each a function of \( w_i \) only. Note that all of these three constraints are affine functions with respect to \( w_i \).

Constraint (C.4) becomes the following affine constraint after renaming:

\[ 1 \leq f_i \times t_{i}^{rem} \quad \forall i \in E_j \]

Observe that constraint (C.5) in the original problem is not affine as the decision variables \( w_i \) and \( t_i \) are multiplied by each other. However, after renaming the variables,
the constraint is now transformed into the following equivalent linear (and thus affine) constraint:

\[ w_i = f_i \times v_i^{rem} \quad \forall i \in E_j \cup I_j \]

As a result, by letting \( t_i = 1/f_i \), the original problem transforms into a convex optimization problem as the new objective function \(- (R_j - \mathcal{X}_j)\) is convex that we seek to minimize and as all equality and inequality constraints of the new equivalent problem are now affine. The optimal solution for such problems can be found easily and quickly using convex optimization solvers such as CVX [58], which is the one used in our implementation.

3.5 Framework Evaluation

The experimental evaluations presented in this section are based on real traces [15] that include all the tasks that were submitted to a Google cluster that is made up of more than 12K PMs. The PMs within that cluster have three types in terms of their supported CPU capacity. Table 3.1 shows the number of PMs for each one of these types along with their CPU capacities normalized with respect to the PM with the largest capacity in the cluster. Since the size of the Google traces is huge (\( \approx 39 \) GB), we limit our evaluations to a chunk of the traces that spans 30 hours.

<table>
<thead>
<tr>
<th>Number of PMs</th>
<th>CPU Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>798</td>
<td>1.00</td>
</tr>
<tr>
<td>11658</td>
<td>0.50</td>
</tr>
<tr>
<td>126</td>
<td>0.25</td>
</tr>
</tbody>
</table>

For each task request \( i \) found in the traces, Google reports:
• a timestamp that indicates when the task was submitted.

• $w_i^{\text{trace}}$ the task’s reserved amount of CPU resources.

• $t_i^{\text{trace}}$ the duration after which the task was accomplished based on the reserved resources.

Unfortunately the traces do not reveal the type or the nature of the submitted tasks and thus we could not infer the elastic tasks from the inelastic ones. In our evaluations, $\rho$ percent of the tasks found in the traces are picked at random and are assumed to be elastic. We use the information revealed by the traces to determine the requested demands of these tasks. For each task $i$ in the traces that is treated as inelastic, the requested amount of computing resources $w_i^{\text{req}}$ and the duration for which these resources are needed $t_i^{\text{req}}$ are taken from the trace numbers and are set to be equal to $w_i^{\text{trace}}$ and $t_i^{\text{trace}}$ respectively. For each task $i$ in the traces that is treated as elastic, the requested computing volume $v_i^{\text{req}}$ is calculated from the traces numbers to be $v_i^{\text{req}} = w_i^{\text{trace}} \times t_i^{\text{trace}}$. The duration $t_i^{\text{max}}$ within which the elastic task must be accomplished is set to $t_i^{\text{trace}}$ (i.e., the elastic tasks have a worst case accomplishment time equal to the one reported in the traces). Finally, we had to make assumptions about $w_i^{\text{max}}$, the maximum amount of resources that can be allocated to the elastic task $i$, as no information is revealed about the nature of these tasks. In our experiments, $w_i^{\text{max}}$ is set to: $w_i^{\text{max}} = w_i^{\text{trace}} + \lambda \times w_i^{\text{trace}}$ where $\lambda$ is the elasticity factor. This makes $w_i^{\text{max}}$ proportional to $w_i^{\text{trace}}$ where the assumption here that the higher the reserved resources for the tasks in the traces, the higher the maximum amount of resources that these tasks can be allocated.

Our proposed framework is compared against the following resource allocation schemes:

• **BF Min:** the BF heuristic [26] is used to make PM placement decisions where a submitted request is placed on a PM in sleep only if it can’t be fitted in any ON
PM and the PM with the largest capacity is chosen in that case for placement. If multiple ON PMs can fit the task, then the one with the least slack is selected for placement. The minimum amount of resources $w^{\text{min}}$ is allocated for the elastic tasks throughout their lifetimes so that they finish exactly in $t^{\text{max}}$ period.

- **BF Max**: the BF heuristic is used to make PM placement decisions, and the maximum amount of the task’s supported resources $w^{\text{max}}$ are allocated to the elastic tasks throughout their lifetimes so that they are accomplished as fast as possible.

- **BF Rand**: the BF heuristic is used to make PM placement decisions, and for each requested elastic task $i$, a random value between $w^{\text{min}}_i$ and $w^{\text{max}}_i$ is allocated to that task throughout its whole lifetime. It is worth mentioning that any selected value within that range guarantees that the task finishes within $t^{\text{max}}_i$ period.

The conducted experiments are organized into two subsections based on the type of service that the elastic tasks request.

### 3.5.1 Best-Effort Service

We consider first the case where all the elastic tasks request a best-effort service. This means that clients are only interested in getting their elastic tasks completed before the specified deadlines and will not be charged more for getting their tasks completed earlier than the deadlines. Our framework’s resource management module in that case tunes the allocated resources in a way that minimizes the PMs energy expenses (by minimizing the PMs uptime) as the PMs revenues are now constant since there are no delay-sensitive elastic tasks.
In the following comparative studies, we consider first the case where $\rho = 50\%$ and $\lambda = 0.5$ and provide a detailed comparison of the different schemes in terms of number of active PMs, cluster utilization and cluster energy expenses. We then vary $\rho$ and $\lambda$ and report the energy expenses of the different schemes under the different experiment setups.

3.5.1.1 Number of Active PMs

We analyze first the number of ON PMs that were needed to serve the submitted tasks as this number has a direct impact on the cluster’s energy expenses. Fig. 3.4 shows the number of ON PMs in the Google cluster over time when different schemes were used to manage the traces tasks. Observe that the BF Max scheme used the largest number of ON PMs most of the time. This is due to the fact that by allocating the maximum amount of resources for the elastic tasks, it is true that these tasks were released as quickly as possible, however, they required a large amount of CPU resources all the time and thus the scheme faced many instances where it was forced to switch new PMs ON as there was no enough slack on any of the currently ON PMs to fit the coming requests. The BF Rand allocation scheme had a similar performance where many ON PMs were also needed to serve the submitted task requests. This clearly shows that allocating random amount of resources to the elastic tasks does not lead into efficient use of the cloud resources. Observe that the BF Min allocation scheme used lesser number of PMs most of the time. Although elastic tasks in that case took longer time to be released, they were allocated smaller amounts of CPU resources leaving larger slacks to support the coming tasks which reduced the need for switching new PMs from sleep. Observe that our framework used the least number of ON PMs all the time compared to all the
other schemes. Both the VM placement and the resource management modules were making efficient placement and tuning decisions in order to reduce both the number of ON PMs and the duration for which PMs need to be kept ON. It is worth mentioning that the number of ON PMs for any of the four schemes shown in Fig. 3.4 exhibits some temporal fluctuations due to the variation in the number and in the resource demands of the tasks that were requested by the clients over time.
3.5.1.2 Utilization Gains

We compare next the utilization gains that are achieved by the different resource allocation schemes where the CPU utilization of a PM (referred to by $\eta$) is the aggregate amount of CPU resources reserved for all of its hosted VMs divided by the PM’s capacity. Fig. 3.5 shows the average utilization for all of the ON PMs in the cluster over time under the different resource allocation schemes. Observe that the BF Max and the BF Rand allocation schemes had the worst average utilization. This clearly shows that not only those two schemes used a large number of ON PMs, but also the resources of those ON PMs were not utilized efficiently. Although the BF Min scheme allocated the least amount of resources for the elastic tasks, it had an improved average utilization over time compared to those schemes as it used less number of ON PMs. Finally, our framework achieved the highest average utilization among all the other schemes. In fact, our framework reached in some cases an average utilization level that is very close to 100%. This is attributed to the VM placement module which packs the submitted tasks tightly and to the resource management module which reduced the wasted resource slacks by increasing the amount of allocated resources for the elastic tasks whenever possible.
3.5.1.3 Energy Savings

We assess next the energy savings that our framework achieves. Experiments in [14] show that the consumed power, $P_{on}$, of an active PM increases linearly from $P_{active}$ to $P_{peak}$ as its CPU utilization, $\eta$, increases from 0 to 100%. More specifically, $P_{on}(\eta) = P_{active} + \eta(P_{peak} - P_{active})$, where $P_{peak} = 400$ and $P_{active} = 200$ Watts. On the other hand, a PM in the sleep state consumes about $P_{sleep} = 100$ Watts. Switching a PM from sleep to ON consumes $E_{s\rightarrow o} = 4260$ Joules, whereas switching a PM from ON to sleep consumes $E_{o\rightarrow s} = 5510$ Joules. All of these numbers are based on real servers’ specifications [30] and are used throughout all the energy evaluations that are mentioned in the chapter.

We calculate the total energy to run the cluster under the different resource allocation schemes where the total energy is the summation of the energy consumed by both ON and sleep PMs in addition to the switching energy (from sleep to ON and vice versa). Fig. 3.6 shows the total energy consumed by the cluster throughout the whole 30-hour traces period normalized with respect to the total energy cost of BF Max scheme. Observe that our proposed framework met the demands of all of the requests within the traces while consuming the least amount of energy compared to all other resource allocation schemes. This proves the efficiency of our framework and highlights how important it is to make efficient placement and resource tuning decisions in cloud centers. It is worth mentioning that the energy expenses (in dollars) for running the cluster under the different schemes are directly proportional to the energy consumption numbers reported in Fig. 3.6 as one merely needs to multiply the energy consumption numbers by the electricity price (given in dollars per energy unit) to get the corresponding expenses for these schemes.

All the previous experiments where for the case when $\rho = 50\%$ and $\lambda = 0.5$. For
completeness, we compare the energy consumption of the different resource management schemes under different $\rho$ and $\lambda$ values. Fig. 3.7 plots the cluster’s total energy consumption (in MegaWatt hour) for the different schemes under different values of $\rho$ when $\lambda$ is fixed to be 0.5. Observe that our proposed framework consumes the least energy and thus has the least expenses when compared to all the other schemes for the different values of $\rho$. Fig. 3.8 also shows the cluster’s total energy (in MegaWatt hour) during the 30 hour trace period for the different schemes when $\rho$ is fixed to be 100% and under different values of elasticity factor $\lambda$. Results show also that our framework had lower energy consumption than all the remaining schemes for the different values of $\lambda$.
3.5.2 Best-Effort and Delay-Sensitive Services

We now consider the case when the elastic tasks are a mix of best-effort and delay sensitive requests, and analyze the energy expenses, the collected revenues and the profits of the different resource allocation schemes under such case. In the following experiments, $\rho = 50\%$, meaning that half of the tasks found in the traces are assumed to be inelastic whereas the remaining ones are elastic, where the elasticity factor is $\lambda = 0.5$. These elastic tasks are split in half at random between best-effort and delay-sensitive requests. Table 3.2 summarizes the values that were selected for the different parameters in our comparative studies. We tried our best to rely on real values when selecting those parameters where $\xi$ was selected based on the average price of electricity in U.S. during 2014 [51], whereas the inelastic task’s cost per volume $\varphi$ is based on the pricing of Microsoft’s cloud computational service as of March, 2015 [59]. We made assumptions regarding the reduced price for the elastic service $\phi$ where it was set 10% less than the volume price for the inelastic service $\varphi$. Finally, the extra charge for completing a delay-sensitive elastic task earlier is 0.25 $/ \text{hour}$.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electricity Price</td>
<td>$\xi$</td>
<td>0.07</td>
<td>$/ \text{kWh}$</td>
</tr>
<tr>
<td>Inelastic task price per volume</td>
<td>$\varphi$</td>
<td>4.9</td>
<td>$/ \text{CPU hour}$</td>
</tr>
<tr>
<td>Elastic task price per volume</td>
<td>$\phi$</td>
<td>4.41</td>
<td>$/ \text{CPU hour}$</td>
</tr>
<tr>
<td>Extra charge for early completion</td>
<td>$\delta$</td>
<td>0.25</td>
<td>$/ \text{hour}$</td>
</tr>
</tbody>
</table>
3.5.2.1 Collected Revenues

We start first by comparing in Fig. 3.9 the total revenues collected from all the served tasks that the Google cluster received during the 30-hour testing period, where the results are normalized with respect to BF Max revenues. Observe that the BF Max scheme had the highest collected revenues among all remaining schemes. This is anticipated as this scheme allocates the maximum amount of resources for the elastic tasks all the time so that they finish as fast as possible. Thus delay-sensitive tasks were charged the highest cost since they were completed as fast as possible, leading into the maximal revenues. Observe that our proposed framework had the second highest revenues as the collected revenues were part of the objective of the optimization problem that our resource management module seeks to maximize. Observe also that the BF Min scheme had the least revenues. This is also expected as this scheme allocates the least amount of resources for the elastic tasks all the time so that they finish exactly in their maximal tolerable periods. Thus the delay-sensitive elastic tasks were charged in that case the least cost based only on their volumes as no extra charges were collected for early completion.
3.5.2.2 Energy Expenses

We report next in Fig. 3.10 the total energy expenses for running the Google cluster under the different schemes normalized with respect to the BF Max expenses. Observe that our proposed framework had the least expenses among all the different schemes. This is attributed to the placement module which places the received requests in a way that minimizes the number of ON PMs and the duration for which PMs are kept ON, and to the resource management module which considers the expenses when tuning the allocated resources for the hosted tasks. The remaining schemes kept a larger number of ON PMs for longer periods which increased significantly their expenses.

3.5.2.3 Profits

Finally, we report in Fig. 3.11 the total profits (the difference between the revenues and the expenses) for the different schemes normalized with respect to our framework’s profits. Observe that our framework had the highest profits when compared to all the other schemes. More specifically, our framework had 25%, 29% and 40% higher profits than the BF Max, the BF Rand and the BF Min resource management schemes. This is attributed to the fact that our framework had the least expenses (as was shown in Fig. 3.10) and had high revenues (as was shown in Fig. 3.9). It is worth mentioning that our framework was also compared against the remaining schemes when considering different parameter values (different pricing values, different percentages of best-effort and delay sensitive tasks, etc.). In all those cases, our framework had higher profits compared to all the remaining schemes.
3.6 Conclusions

We propose in this chapter a profit-driven online resource allocation framework for elastic and inelastic task requests. Our framework exploits the elasticity and the varying charging costs among the submitted requests and decides where to place the heterogeneous submitted task requests, and how much resources should be allocated to the elastic ones such that the cloud profits are maximized while meeting all tasks demands. Comparative evaluations based on Google traces showed that our framework increased the cloud profits significantly by considering both the energy expenses and the collected revenues in its allocation decisions.

3.7 Future Work Directions

We end the chapter by providing some directions on cloud resource management that require further exploration in future.

a) Tasks with Multiple Resources: The focus of this chapter was on the case where tasks (elastic/inelastic) have certain CPU resource demands. An interesting direction for future work would be to consider the case when these heterogeneous tasks have other resource demands in addition to CPU (e.g. memory and bandwidth). Managing resources becomes more complex in that case as the resource manager needs to account for the interaction and for the relations among the different resources.

b) Tasks with Dependencies: Another open problem is how to schedule and allocate resources in a profitable way when there are dependencies among the tasks. A submitted task in that case might be dependent on the output of other tasks and thus can only be scheduled after the completion of those tasks.
c) **Task Pricing Models:** We believe that further efforts should be made by the industry and by the research community to adapt and develop pricing models for the heterogeneous tasks. There are many questions that require further investigation such as by how much should the elastic services be cheaper than the inelastic ones? How much should the extra charge for completing a delay-sensitive elastic task earlier be? and whether or not to consider non-linear charging models\(^1\) for the delay-sensitive service.

d) **Testing on Further Real Traces:** Finally, the lack of public release of real traces similar to those provided by Google has prevented us from testing our framework on other traces. We hope to have the chance to do that in future.

\(^1\). A nice property of the optimization problem solved by our proposed resource management module is the fact that the problem remains convex for any charging model (even non-linear ones) as long as the charging cost is a convex function of the task's completion time.
Chapter 4: Energy-Efficient Resource Overcommitment Framework

We propose in this chapter an integrated, energy-efficient, resource allocation framework for overcommitted clouds. The framework makes great energy savings by 1) minimizing Physical Machine (PM) overload occurrences via VM resource usage monitoring and prediction, and 2) reducing the number of active PMs via efficient VM migration and placement. Using real Google data consisting of a 29-day traces collected from a cluster containing more than 12K PMs, we show that our proposed framework outperforms existing overload avoidance techniques and prior VM migration strategies by reducing the number of unpredicted overloads, minimizing migration overhead, increasing resource utilization, and reducing cloud energy consumption.

4.1 **INTRODUCTION**

Reducing the energy consumption of datacenters has received a great attention from the academia and the industry recently [60, 61]. Recent studies indicate that datacenter servers operate, most of the time, at between 10% and 50% of their maximal utilizations. These same studies, on the other hand, also show that servers that are kept ON but are idle or lightly utilized consume significant amounts of energy, due to the fact that an idle ON server consumes more than 50% of its peak power [14]. It can therefore be concluded that in order to minimize energy consumption of datacenters, one needs to consolidate cloud workloads into as few servers as possible.

Upon receiving a client request, the cloud scheduler creates a virtual machine (VM),
allocates to it the exact amounts of CPU and memory resources requested by the client, and assigns it to one of the cluster’s physical machines (PMs). In current cloud resource allocation methods, these allocated resources are reserved for the entire lifetime of the VM and are released only when the VM completes. A key question, constituting the basis for our work motivation, that arises now is to see whether the VMs do utilize their requested/reserved resources fully, and if not, what percentage of the reserved resources is actually being utilized by the VMs. To answer this question, we conduct some measurements on real Google traces\(^1\) [47] and show in Fig. 4.1 a one-day snapshot of this percentage. Observe that only about 35% of the VMs’ requested CPU resources and only about 55% of the VMs’ requested memory resources are actually utilized by the VMs. Our study clearly indicates that cloud resources tend to be overly reserved, leading to substantial CPU and memory resource wastage. Two main reasons are behind this over-reservation tendency: First, cloud clients usually do not know the exact amounts of resources their applications would need, so they tend to overestimate them in order to guarantee a safe execution. Second, due to and depending on the nature of the applications hosted on the PMs, the level of utilization of the requested resources may change over time and may even rarely reach its peak, making it impossible for the VM to use the full amount of its requested resources.

Resource overcommitment [62, 63] is a technique that has been recognized as a potential solution for addressing the above-mentioned wastage issues. It essentially consists of allocating VM resources to PMs in excess of their actual capacities, expecting that these actual capacities will not be exceeded since VMs are not likely to utilize their reserved

\(^1\)It is important to mention that Google, as reported in the traces, allocates containers instead of full VMs when handling task requests. However, our framework remains applicable regardless of whether tasks are handled by containers or by full virtualization. Throughout this work, we refer to the task requests submitted to the Google cluster by VM requests.
resources fully. Therefore, it has a great potential for saving energy in cloud centers, as VMs can now be hosted on fewer ON PMs.

Resource overcommitment creates, however, a new problem, PM overloading, which occurs when the aggregate resources demands of the VMs scheduled on some PM does indeed exceed the PM’s capacity. When this occurs, some or all of the VMs running on the overloaded PM will experience performance degradation (some VMs may even crash), possibly leading to violations of service-level agreements (SLAs) between the cloud and its clients. Although VM migration [64] can be used to handle PM overloads, where some of the VMs hosted by the overloaded PM are moved to other under-utilized or idle PMs, it raises two key challenges/questions:

i) When should VMs be migrated?

ii) Which VMs should be migrated and which PMs these VMs should be migrated to?

The framework described in this section addresses these two challenges. It proposes an integrated resource allocation framework that improves resource utilization, reduces energy consumption, and avoids, as much as possible, SLA violations in cloud datacenters. More specifically, our proposed framework:
• predicts future resource utilizations of scheduled VMs, and uses these predictions to make efficient cloud resource overcommitment decisions to increase utilization.

• predicts PM overload incidents and triggers VM migrations before overloads occur to avoid SLA violations.

• performs energy-efficient VM migration by determining which VMs to migrate and which PMs need to host the migrated VMs such that the migration energy overheads and the number of active PMs are minimized.

The effectiveness of our techniques is evaluated and compared against existing ones by means of real Google traces [47] collected from a heterogeneous cluster made up of more than 12K PMs.

The remainder of the chapter is organized as follows. Section 4.2 discusses prior work. Section 4.3 provides an overview of the proposed framework. Section 4.4 presents our proposed prediction methods. Section 4.5 presents our formulation of the VM migration problem, and Section 4.6 presents a heuristic for solving it. Section 4.7 evaluates the effectiveness of the proposed framework. Finally, Section 4.8 concludes the chapter and provides directions for future work.

4.2 RELATED WORK

We review next how our framework differs from prior overload avoidance and VM migration techniques in overcommitted clouds.
4.2.1 Overload Avoidance Techniques

One approach proposed in [65–67] to handle PM overloads consists essentially of trig-
ergerng VM migrations upon detecting overloads. This stops VMs from contending over
the limited resources, but can clearly result in some SLA violations. To address this
limitation, the techniques presented in [68–74], which are referred to as threshold-based
techniques, propose to trigger migrations as soon as the PM’s utilization exceeds a cer-
tain threshold, but before an overload actually takes place. The assumption here is that
there is a high chance that an overload occurs when the PM’s utilization exceeds the
set threshold. The threshold could be set statically as in VMware [68] and as in [69–
71], where the threshold is typically set to 90% utilization; or dynamically as in [72–74],
where it is tuned for each PM based on how fluctuating the PM’s workload is. The higher
the PM’s utilization fluctuations, the lower the PM’s selected threshold and vice versa.
Although threshold-based techniques reduce overloads, they put a limitation on the uti-
"lization gains that can be achieved as they leave a certain unutilized slack for each PM.
Furthermore, these techniques can trigger many unnecessary migrations as exceeding the
set threshold does not necessary mean that an overload is going to happen.

To tackle these limitations, we propose a prediction-based overload-avoidance tech-
nique that predicts future resource demands for each scheduled VM so that overloads
can be foreseen and migrations can be triggered ahead of time. These predictions also
help us decide where to place the newly submitted VMs so that the utilization gains due
to overcommitment are increased while avoiding overloads as much as possible.

There have been previous attempts to predict the resource demands of cloud applica-
tions [75–77]. Fourier transformation [75] and SVM predictors [76] were used to extract
periodic patterns from the traces of certain cloud applications. The extracted patterns
were used later to predict the resource demands for those applications. MapReduce jobs were profiled in [77] with the aim of predicting the resource demands for newly submitted instances of those jobs. Our prediction module differs from the previously proposed offline predictive-based techniques [75–77] in that it uses a customized adaptive Wiener filter [78, 79] that learns and predicts the resource demands of the clients’ scheduled VMs online without requiring any prior knowledge about the hosted VMs. The closest work to our proposed predictive technique is the work in [80], where the authors developed an online scheme called Fast Up Slow Down (FUSD) that predicts the future demands of the scheduled VMs and that triggers migrations to avoid overloads. The FUSD scheme [80] and the threshold-based overload avoidance techniques [69–71] are used as benchmarks where we show in our evaluations that our proposed prediction technique not only reduces the number of unpredicted overloads, but also achieves greater energy savings when compared to those benchmark techniques.

4.2.2 VM Migration Techniques

Live migration is now available in VMware [68] and Xen [81] where the downtime that the clients experience due to the migration process is extremely low and ranges between tens of milliseconds to a second [22]. Despite the maturity of the VM migration technology, it is still challenging to find efficient resource management strategies that decide what VMs to migrate and what PMs should be the destination for each migration when an overload is encountered.

Largest First heuristics [82–85] addresses this challenge by moving VMs with the largest resource demands to the PMs with the largest resource slacks while trying to minimize the number of needed migrations. Several enhancements were added to this
heuristic in [23, 86] with the aim of placing VMs that often communicate with each other close to one another. The main limitation of all the previous heuristics is that they completely ignore the energy overhead associated with migrations.

The authors in [87, 88] proposed migration policies that take migration cost into account. One of the drawbacks of the approach presented in [87] lies in its limitation to VMs with single resource (CPU). Sandpiper [88] is a multi-resource heuristic that aims at moving the largest amount of resources from the overloaded PM with the least cost. However, both works of [87, 88] share a common limitation as they completely ignore the power state of the PMs in the cluster. Although these approaches may move VMs that have low migration cost, there may be no already-ON PMs that have enough slack to host the migrated VMs. This forces turning PMs from sleep to ON to host the moved VMs, which comes with a high energy cost [55, 56] and increases the number of ON machines in the cloud cluster, leading to larger energy consumption.

Unlike previous works, our proposed framework has an energy-aware migration module that makes migration decisions that minimize the total migration energy overhead, which is made up of the energy spent to move VMs and that to switch PMs ON to host the migrated VMs. Our experimental studies presented in Section 4.7 show that our proposed migration method achieves higher energy savings and consolidates the workload in fewer PMs when compared to both the Largest First and Sandpiper heuristics, discussed above.

4.3 PROPOSED FRAMEWORK

Our proposed framework is suitable for heterogeneous cloud clusters whose PMs may or may not have the same resource capacities. Although our framework is extendable to
any number of resources, in this chapter, we consider two resources: CPU and memory. Thus, a PM $j$ can be represented by $[C_{\text{cpu}}^j, C_{\text{mem}}^j]$, where $C_{\text{cpu}}^j$ and $C_{\text{mem}}^j$ are the PM's CPU and memory capacities. Throughout, let $P$ be the set of all PMs in the cloud cluster. Recall that a client may, at any time, submit a new VM request, say VM $i$, represented by $[R_{\text{cpu}}^i, R_{\text{mem}}^i]$ where $R_{\text{cpu}}^i$ and $R_{\text{mem}}^i$ are the requested amounts of CPU and memory. Whenever the client no longer needs the requested resources, it submits a VM release request. Throughout, let $V$ be the set of all VMs hosted by the cluster.

In this section, we provide a brief overview of the different components of the proposed framework so as to have a global picture of the entire framework before delving into the details. As shown in Fig. 4.2, our framework has the following modules:
4.3.1 Resource Predictor

A separate Resource Predictor module is dedicated to each VM scheduled on the cluster. The Resource Predictor module for VM \( i \) consists of two predictors (one for CPU and one for memory) that monitor and collect the VM’s CPU and memory usage traces, and use them, along with other VM parameter sets (to be learned online from the VM’s resource demands behaviors), to predict the VM’s future CPU and memory demands, \( P_{cpu}^i \) and \( P_{mem}^i \). These predictions are calculated for the coming \( \tau \) period and are done periodically at the end of each period. Detailed description of how predictors work and how these parameters are updated are given in Section 4.4.

4.3.2 Overload Predictor

Each PM in the cluster is dedicated an Overload Predictor module. The Overload Predictor module assigned for PM \( j \) fetches the predicted CPU and memory demands for all the VMs that are hosted on that PM and calculates PM \( j \)’s predicted aggregate CPU and memory demands, \( U_{cpu}^j \) and \( U_{mem}^j \) respectively, as follows:

\[
U_{cpu}^j = \sum_{i \in V : \theta(i) = j} P_{cpu}^i \quad \text{and} \quad U_{mem}^j = \sum_{i \in V : \theta(i) = j} P_{mem}^i
\]

where \( \theta : V \rightarrow P \) is the VM-PM mapping function, with \( \theta(i) = j \) meaning that VM \( i \) is hosted on PM \( j \). The module then compares the calculated aggregate CPU and memory demands, \( U_{cpu}^j \) and \( U_{mem}^j \), with PM \( j \)’s supported CPU and memory capacity, \( C_{cpu}^j \) and \( C_{mem}^j \). If \( U_{cpu}^j > C_{cpu}^j \) or \( U_{mem}^j > C_{mem}^j \), then the Overload Predictor notifies the Cluster Manager that PM \( j \) is expected to have an overload in the coming period.
The Overload Predictor also forwards to the Cluster Manager the predicted CPU and memory demands, \( P_{\text{cpu}}^i \) and \( P_{\text{mem}}^i \), for each VM \( i \) hosted on PM \( j \). These predictions will be used by the Energy-Aware Migration module to decide what VM(s) to keep, what VM(s) to migrate and to where the migrated VM(s) should be moved such that the predicted overload is avoided in future. If no overload is predicted on PM \( j \), then the Overload Predictor module forwards to the Cluster Manager the predicted CPU slack and memory slack for PM \( j \), referred to by \( S_{\text{cpu}}^{j} \) and \( S_{\text{mem}}^{j} \) respectively, which are calculated as follows:

\[
S_{\text{cpu}}^{j} = C_{\text{cpu}}^{j} - U_{\text{cpu}}^{j} \quad \text{and} \quad S_{\text{mem}}^{j} = C_{\text{mem}}^{j} - U_{\text{mem}}^{j} \quad (4.1)
\]

The CPU and memory slacks on PM \( j \) need to be known to the Cluster Manager as it may decide to place a VM (a newly submitted VM or a migrated VM) on PM \( j \) and thus the Cluster Manager needs to make sure that enough slacks are available on the PM to support the resource requirements of the newly placed VM.

As expected with any prediction framework, it is also possible that our predictors fail to predict an overload. We refer to such incidents as *unpredicted overloads*, which will be eventually detected when they occur as the Overload Predictor also tracks how much resources all the VMs scheduled on the PM are actually using over time. For any predicted PM overload, VM migration will be performed before the overload actually occurs, thus avoiding it. But for each unpredicted PM overload, VM migration will be performed upon its detection. All VM migrations are handled by the Energy-Aware VM Migration sub module in the Cluster Manager.
4.3.3 Cluster Manager

This module is connected to the Overload Predictor modules of each PM in the cluster and is made up of two sub modules:

4.3.3.1 Energy-Aware VM Migration

Let $O_{pm}$ be the set of all PMs that are predicted to have an overload. This sub module determines which VM(s) among those hosted on the PMs in $O_{pm}$ need to be migrated so as to keep the predicted aggregate CPU and memory demands below the PM’s capacity. To make efficient decisions, the module needs to know the energy costs for moving each VM among those hosted on the PMs in $O_{pm}$. We use the notation $m_i$ to refer to the cost (in Joules) for moving the $i^{th}$ VM. This module also determines which PM each migrating VM needs to migrate to. Such a PM must have enough CPU and memory slack to accommodate the migrated VM(s), and thus the module needs to know the available resource slacks on all the PMs in the cluster that are not predicted to have an overload. These are provided to the module as input (through the Overload Prediction module) in addition to the ON-sleep states of the PMs, $\gamma$, where the state function $\gamma(j)$ returns PM $j$’s power state (ON or sleep) prior to migration. After performing the required migrations, the VM-PM mapping $\theta$ and the ON-sleep PM state $\gamma$ get updated. Details on how the the migration problem is formulated and how it is solved by our module are provided in Sections 4.5 and 4.6.
4.3.3.2 Scheduler

This sub module decides where to place newly submitted VMs and also handles VM release events.

The new VM placements are handled with two objectives in mind: saving energy and minimizing the PM overload occurrence probability. When the client requests a VM, the Scheduler places the VM request on a sleeping PM only if no ON PM is predicted to have enough CPU slack ($S_{cpu}$) and enough memory slack ($S_{mem}$) to provide the requested CPU and memory resources of the submitted VM request. The sleeping PM with the largest capacity ($C_{cpu} \times C_{mem}$) in that case is switched ON and is selected to host the submitted VM request. On the other hand, if multiple ON PMs have enough predicted CPU and memory slacks to host the submitted VM request, then the predicted slack metric (defined for a PM $j$ as $S_{cpu,mem}^j = S_{cpu}^j \times S_{mem}^j$) is calculated for each one of those PMs. The ON PM with the largest predicted slack metric among those that can fit the VM request is selected to host the submitted VM.

The intuition behind our placement policy is as follows: It is better to host a newly submitted VM request on an ON PM, so as to avoid wakening up asleep machines. This saves energy. If there is no option but to place the VM request on a sleeping PM, then the request is placed on the sleeping PM that has the largest capacity as this PM can fit a larger number of VM requests that might be received next, which reduces the chances of waking up another sleeping PM in future. On the other hand, if multiple ON PMs can fit the submitted VM request, then the one with the largest predicted slack is picked to host the submitted VM request so as to decrease the chances of having an overload after that placement. The product of the CPU and memory resources is used to combine the server’s multiple resources into a single sorting metric. This metric is used to decide
which sleeping PM has the largest capacity or which ON PM has the largest predicted slack when comparing two PMs.\footnote{It is worth mentioning that there exists metrics other than the product that can be used to combine the multiple resources. The summation of the CPU and memory resources can be used for example as a combining metric. In our framework, we selected the product metric over the summation metric as experiments in our prior work \cite{56} showed that for the Google traces, the product metric makes slightly more compact VM placements when compared to the summation metric. The other metrics mentioned in \cite{89} such as the Dot Product and the Norm-Based Greedy were not considered in our framework due to their high computational overhead.}

A Resource Predictor module is constructed for the new VM after assigning it to a PM. The Resource Predictor module consists of two predictors (one for CPU and one for memory) and will be used to monitor the resource demands of the VM and to make future demand predictions, as described earlier.

Upon receiving a VM release from the client, the Scheduler releases the VM's allocated CPU and memory resources, frees all system parameters associated with the VM (e.g., predictors), and updates the aggregate CPU and memory predictions of the hosting PM accordingly. The PM is switched to sleep to save energy if it becomes vacant after releasing the VM.

4.4 VM Resource Predictor

We explain in this section how a predictor for a scheduled VM predicts its future resource demands in the coming $\tau$ minutes, where the term resource will be used to refer to either CPU or memory. In our framework, we choose to use the Wiener filter prediction approach \cite{78,79} for several reasons. First, it is simple and intuitive, as the predicted utilization is a weighted sum of the recently observed utilization samples. Second, prediction weights can easily be updated without requiring heavy calculations or large storage space. Finally, it performs well on real traces as will be seen later.
Let $n$ be the time at which resource predictions for a VM need to be made. The following notations will be used throughout:

- $z[n - i]$: is the VM’s average resource utilization during period $[n - (i + 1)\tau, n - i\tau]$ minutes.
- $d[n]$: is the VM’s actual average resource utilization during period $[n, n + \tau]$.
- $\hat{d}[n]$: is the VM’s predicted average resource utilization during period $[n, n + \tau]$.

Wiener filters predict resource utilizations while assuming wide-sense stationarity of $z[n]$. The predicted average resource utilization, $\hat{d}[n]$, is a weighted average over the $L$ most recently observed utilization samples; i.e., $\hat{d}[n] = \sum_{i=0}^{L-1} w_i z[n - i]$, where $w_i$ is the $i^{th}$ sample weight. The prediction error, $e[n]$, is then the difference between the actual and predicted utilizations; i.e., $e[n] = d[n] - \hat{d}[n] = d[n] - \sum_{i=0}^{L-1} w_i z[n - i]$. The objective is to find the weights that minimize the Mean Squared Error ($MSE$) of the training data, where $MSE = E[e^2[n]]$. Differentiating $MSE$ with respect to $w_k$ and setting this derivative to zero yields, after some algebraic simplifications, $E[d[n]z[n - k]] - \sum_{i=0}^{L-1} w_i E[z[n - k]z[n - i]] = 0$. It then follows that $r_{dz}(k) = \sum_{i=0}^{L-1} w_i r_{zz}(i - k)$ where

$$r_{dz}(k) = E[d[n]z[n - k]] \tag{4.2}$$

$$r_{zz}(i - k) = E[z[n - k]z[n - i]] \tag{4.3}$$

Similar equations expressing the other weights can also be obtained in the same way.
These equations can be presented in a matrix format as $R_{dz} = R_{zz} W$, where

$$R_{zz} = \begin{bmatrix} r_{zz}(0) & r_{zz}(1) & \ldots & r_{zz}(L-1) \\ r_{zz}(1) & r_{zz}(0) & \ldots & r_{zz}(L-2) \\ \vdots & \vdots & \ddots & \vdots \\ r_{zz}(L-1) & r_{zz}(L-2) & \ldots & r_{zz}(0) \end{bmatrix}$$

$$W = \begin{bmatrix} w_0 & w_1 & \ldots & w_{L-1} \end{bmatrix}^T$$

$$R_{dz} = \begin{bmatrix} r_{dz}(0) & r_{dz}(1) & \ldots & r_{dz}(L-1) \end{bmatrix}^T$$

Given $R_{zz}$ and $R_{dz}$, the weights can then be calculated as:

$$W = R_{zz}^{-1} R_{dz} \quad (4.4)$$

The elements of $R_{zz}$ are calculated using the unbiased correlation estimation as:

$$r_{zz}(i) = \frac{1}{N-i} \sum_{j=0}^{N-i-1} z[j+i] z[j] \quad (4.5)$$

where $N$ is the VM’s number of observed samples with each sample representing an average utilization over $\tau$ minutes.

The elements of $R_{dz}$ can also be estimated using the correlation coefficients. Since $d[n]$ represents the average resource utilization in the coming $\tau$ minutes, we can write $d[n] = z[n + 1]$. Plugging the expression of $d[n]$ in Eq. (4.2) yields $r_{dz}(k) = E[z[n + 1]z[n-k]] = r_{zz}(k+1)$, and thus $R_{dz} = \begin{bmatrix} r_{zz}(1) & r_{zz}(2) & \ldots & r_{zz}(L) \end{bmatrix}^T$. The elements of $R_{dz}$ can be calculated using Eq. (4.5). An MSE estimation of the weight vector follows
then provided $R_{dz}$ and $R_{zz}$.

**Adaptive Updates** Note that $N = L + 1$ samples need to be observed in order to calculate the predictor’s weights. When the number of samples available during the early period is less than $L + 1$, no prediction will be made and we assume that VMs will utilize all of their requested resources. Once the predictor observes $N = L + 1$ samples, the correlations $r_{zz}(i)$ can then be calculated for all $i$ allowing the weights to be estimated.

When $N > L+1$, the predictor adapts to new changes by observing the new utilization samples, updates the correlations, and calculates the new updated weights. This results in increasing the accuracy of the predictor over time, as the weights are to be calculated based on a larger training data. From Eq. (4.5), the coefficient $r_{zz}(i)$ can be written as $Sum(i)/Counter(i)$, where $Sum(i) = \sum_{j=0}^{N-i-1} z[j+i]z[j]$ and $Counter(i) = N - i$ are two aggregate variables. Now recall that every $\tau$ minutes, a new resource utilization sample $z[k]$ is observed, and hence, the aggregate variables can be updated as $Sum(i) \leftarrow Sum(i) + z[k]z[k-i]$ and $Counter(i) \leftarrow Counter(i) + 1$ and the correlation $r_{zz}(i)$ is updated again as $Sum(i)/Counter(i)$. The updated weights are then calculated using Eq. (4.4), which will be used to predict the VM’s future resource utilizations. Note that only two variables need to be stored to calculate $r_{zz}$ instead of storing all the previous traces, and thus the amount of storage needed to update these weights is reduced significantly.

**Safety Margin** Our stochastic predictor may still make errors by over- or under-estimating resource utilizations. When under-estimation occurs, overcommitted PMs may experience overloads, potentially leading to some performance degradation. In order to take an even more conservative approach towards reducing such occurrences,
we add a safety margin. We consider an adaptive approach for setting safety margins, where margin values are adjusted based on the accuracy of the predictor. Essentially, they depend on the deviation of the predicted demands from the actual ones; i.e., the higher the deviation, the greater the safety margin. In our framework, we calculate an exponentially weighted moving average (EWMA) of the deviation, $Dev$, while giving higher weights to the most recent ones. Initially, we set $Dev[0] = 0$, and for later time $n$, we set $Dev[n] = (1 - \alpha)Dev[n-1] + \alpha |d[n-1] - \hat{d}[n-1]|$, where $0 < \alpha < 1$ is the typical EWMA weight factor used to tune/adjust the weight given to most recent deviations. The predicted average utilization with safety margin, $\bar{d}[n]$, can then be calculated as $\bar{d}[n] = \hat{d}[n] + Dev[n]$. Recall that each VM requests two resources: CPU and memory. Hence, two predictions $\tilde{d}_{cpu}[n]$ and $\tilde{d}_{mem}[n]$ are calculated as described above for each VM $v$. Putting it all together, VM $v$’s predicted average CPU and memory utilizations, $P_{cpu}^v$ and $P_{mem}^v$, are calculated as

$$P_{cpu}^v = \begin{cases} R_{cpu}^v & N < L + 1 \\ \min(\tilde{d}_{cpu}[n], R_{cpu}^v) & \text{otherwise} \end{cases}$$

$$P_{mem}^v = \begin{cases} R_{mem}^v & N < L + 1 \\ \min(\tilde{d}_{mem}[n], R_{mem}^v) & \text{otherwise} \end{cases}$$

4.5 ENERGY-AWARE VM MIGRATION

VM migration must be performed when an overload is predicted in order to avoid SLA violations. Since energy consumption is our primal concern, we formulate the problem of deciding which VMs to migrate and which PMs to migrate to as an optimization problem
with the objective of minimizing the migration energy overhead as described next.

**Decision Variables.** Let $O_{vm}$ be the set of VMs that are currently hosted on all the PMs that are predicted to overload in $O_{pm}$. For each VM $i \in O_{vm}$ and each PM $j \in P$, we define a binary decision variable $x_{ij}$ where $x_{ij} = 1$ if VM $i$ is assigned to PM $j$ after migration, and $x_{ij} = 0$ otherwise. Also, for each $j \in P$, we define $y_j = 1$ if at least one VM is assigned to PM $j$ after migration, and $y_j = 0$ otherwise.

**Objective Function.** Our objective is to minimize VM migration energy overhead, which can be expressed as

$$
\sum_{i \in O_{vm}} \sum_{j \in P} x_{ij} a_{ij} + \sum_{j \in P} y_j b_j
$$

(4.6)

and is constituted of two components: **VM moving energy overhead** and **PM switching energy overhead**. VM moving energy overhead, captured by the left-hand summation term, represents the energy costs (in Joule) associated with moving VMs from overloaded PMs. The constant $a_{ij}$ represents VM $i$'s moving cost, and is equal to $m_i$ when VM $i$ is moved to a PM $j$ different from its current PM, and equal to 0 when VM $i$ is left on the same PM where it has already been hosted. Formally, $a_{ij} = 0$ if, before migration, $\theta(i) = j$, and $a_{ij} = m_i$ otherwise. Here $m_i$ denotes VM $i$'s moving energy overhead.

PM switching energy overhead, captured by the right-hand term of the objective function (Eq. (4.6)), represents the energy cost associated with switching PMs from sleep to ON to host the migrated VMs. The constant $b_j = 0$ if PM $j$ has already been ON before migration (i.e., $\gamma(j) = ON$ before migration), and $b_j = E_{s\to o}$ otherwise, where $E_{s\to o}$ is the transition energy consumed when switching a PM from sleep to ON.
**Constraints:** The optimization problem is subject to the following constraints. One,

\[ \sum_{j \in P} x_{ij} = 1 \quad \forall i \in O_{vm} \]

dictating that every VM must be assigned to only one PM. Two,

\[ \sum_{i \in O_{vm}} x_{ij} P_{cpu}^i \leq C_{cpu}^j \quad \forall j \in O_{pm} \]
\[ \sum_{i \in O_{vm}} x_{ij} P_{mem}^i \leq C_{mem}^j \quad \forall j \in O_{pm} \]

which state that the predicted CPU and memory usage of the scheduled VMs on any overloaded PM must not exceed the PM’s available CPU and memory capacities. Three,

\[ \sum_{i \in O_{vm}} x_{ij} P_{cpu}^i \leq S_{cpu}^j \quad \forall j \in PnO_{pm} \]
\[ \sum_{i \in O_{vm}} x_{ij} P_{mem}^i \leq S_{mem}^j \quad \forall j \in PnO_{pm} \]

where \( PnO_{pm} \) is the set of PMs predicted not to be overloaded. \( S_{cpu}^j \) and \( S_{mem}^j \) are the predicted CPU and memory slacks for PM \( j \) calculated using Eq.(4.1). Recall that some VMs will be migrated to PMs that already have some scheduled VMs, and the above constraints ensure that there will be enough resource slack to host any of the migrated VMs. Four,

\[ \sum_{i \in O_{vm}} x_{ij} \leq |O_{vm}| y_j \quad \forall j \in P \tag{4.7} \]

which forces \( y_j \) to be one (i.e., PM \( j \) needs to be ON) if one or more VMs in \( O_{vm} \) will be assigned to PM \( j \) after migration.
Note that if none of the VMs in $O_{vm}$ is assigned to PM $j$, then the constraint (4.7) can still hold even when $y_j$ takes on the value one. In order to force $y_j$ to be zero when no VM is assigned to PM $j$ (i.e. PM $j$ maintains the same power state that it had prior to migration as no VM will be migrated to it), we add the following constraint. Five,

$$1 + \sum_{i \in O_{vm}} x_{ij} > y_j \quad \forall j \in P$$

(4.8)

Note that if one or more VMs is assigned to PM $j$, constraint (4.8) does not force $y_j = 1$ either, but constraint (4.7) does. Thus, constraints (4.7) and (4.8), together, imply that $y_j = 1$ if and only if one or more VMs are assigned to PM $j$ after migration.

After solving the above problem, the optimal $y_j$s indicate whether new PMs need to be turned ON (also reflected via the $\gamma$ function), and the optimal $x_{ij}$s indicate whether new VM-PM mappings are needed (also reflected via the $\theta$ function).

### 4.6 Proposed Heuristic

In the previous section, we formulated the VM migration problem as an integer linear program (ILP). The limitation of this formulation lies in its complexity, arising from the integer variables, as well as the large numbers of PMs and VMs. To overcome this complexity, we instead propose to solve this problem using the following proposed fast heuristic.

Instead of deciding where to place the VMs that are currently hosted on all the overloaded PMs, our proposed heuristic (shown in Algorithm 2) takes only one overloaded PM $P_o$ at a time (line 1), and solves a smaller optimization problem to decide where to place the VMs that are currently hosted on the picked PM $P_o$. We refer to these VMs
that are hosted on $P_o$ prior to migration by $O_s$ (line 2). Another feature adopted by
our heuristic that reduces the complexity further is to consider only a set of $N_{on}$ ON
PMs and $N_{sleep}$ asleep PMs as destination candidates for VM migrations. The set of
selected ON PMs, denoted by $P_{on}$, is formed by the function $\text{PickOnPMs}$ in line 3. The
returned PMs by this function are the ON PMs that have the largest predicted slack
metric $(S_{cpu} \times S_{mem})$. The intuition here is that the PM with the largest predicted slack
has higher chances for having enough space to host the VMs that need to be migrated.
Furthermore, moving VMs to these PMs has the lowest probability to trigger an overload
on these PMs. The set of selected PMs that are asleep is denoted by $P_{sleep}$ and formed
using the function $\text{PickSleepPMs}$ in line 4. The returned PMs are the ones that are
asleep and that have the largest capacity $(C_{cpu} \times C_{mem})$. Again the intuition here is
that PMs with larger capacity have larger space to host the migrated VMs and hence,
the lowest probability of causing an overload.

The heuristic then forms the set $P_s$ (line 5) which is made up of the picked overloaded
PM, $P_o$, the selected ON PMs, $P_{on}$, and the selected sleep PMs, $P_{sleep}$. An optimization
problem similar to the one explained in the previous section is solved with the only
exception that $P = P_s$ and $O_{vm} = O_s$. Solving the optimization problem determines
which VMs in $O_s$ need to be migrated so as to avoid PM overloads. The VMs that are
assigned to one of the ON PMs are then migrated to the assigned ON PMs. As for the
VMs that are assigned to one of the PMs that are in sleep, we try first to place them in
any already ON PMs. To do so, all the ON PMs apart from those in $P_{on}$ are ordered in
a decreasing order of their slacks. The VMs that are assigned to the PMs in sleep are
also ordered from largest to smallest. We iterate over these VMs while trying to fit them
in one of the ordered ON PMs. This is done in order to make sure that no ON PMs
(other than the selected PMs in $N_{on}$) have enough space to host the migrated VMs. If
an already ON PM has enough space to host one of these VMs, then the VM is migrated to the ON PM rather than to the PM in sleep so as to avoid incurring switching energy overhead. Otherwise, the VMs are migrated to the assigned PMs that are asleep, as indicated in line 6.

As for $N_{on}$ and $N_{sleep}$, these parameters affect the size of the optimization problem. The larger the values of these parameters, the higher the number of PM destination candidates, and the longer the time needed to solve the problem but the lower the total migration energy overhead. This point will be further illustrated by the experiments presented in the following section.

\begin{algorithm}
\begin{algorithmic}[1]
\State {\textbf{for} each $P_o \in O_{on}$ \textbf{do}}
\State $O_s = \{ \forall i \in V \text{ s.t. } \theta(i) = P_o \}$
\State $P_{on} \leftarrow \text{PickOnPMs}(N_{on})$
\State $P_{sleep} \leftarrow \text{PickSleepPMs}(N_{sleep})$
\State $P_s \leftarrow P_{on} \cup P_{sleep} \cup P_o$
\State $[\vec{x}, \vec{y}] = \text{SolveOptimization}(P_s, O_s)$
\State {Migrate VMs that should be placed on a PM $\in P_{on}$}
\State {Try placing VMs that should be placed on a PM $\in P_{sleep}$ on any ON PM $\notin P_{on}$}
\State {Update $\theta$ and $\gamma$}
\State {\textbf{end for}}
\end{algorithmic}
\end{algorithm}

\section*{4.7 Framework Evaluation}

The experiments presented in this section are based on real traces of the VM requests submitted to a Google cluster that is made up of more than 12K PMs (see [47] for further details). Since the size of the traces is huge, we limit our analysis to a chunk spanning a 24-hour period. Since the traces do not reveal the energy costs associated with moving the submitted VMs, nor do they reveal enough information that allow us to estimate the costs to move the VMs (e.g. the memory dirtying rate, the network bandwidth,
etc.), we assume that the VM’s moving overhead follows a Gaussian distribution with a mean $\mu = 350 \text{ Joule}$ and a standard deviation $\delta = 100$ [90]. The selection of these numbers is based on the energy measurements reported in [90], which show that the moving overhead varies between 150 and 550 Joules for VM sizes between 250 and 1000 Mega Bytes. These moving costs include the energy consumed by the source PM, the destination PM, and the network links.

As for the power consumed by an active PM, $P(\eta)$, it increases linearly from $P_{\text{idle}}$ to $P_{\text{peak}}$ as its CPU utilization, $\eta = U_{\text{cpu}}/C_{\text{cpu}}$, increases from 0 to 100% [14]. More specifically, $P(\eta) = P_{\text{idle}} + \eta(P_{\text{peak}} - P_{\text{idle}})$, where $P_{\text{peak}} = 400$ and $P_{\text{idle}} = 200$ Watts. A sleeping PM, on the other hand, consumes $P_{\text{sleep}} = 100$ Watts. The energy consumed when switching a PM from sleep to ON is $E_{s\rightarrow o} = 4260$ Joules, and that when switching a PM from ON to sleep is $E_{o\rightarrow s} = 5510$ Joules. These numbers are based on real servers’ specs [91–93].

**Overload Prediction.** We start our evaluations by showing in Fig. 4.3 the number of overloads predicted when our framework is run over the 24-hour trace period. These overload predictions are based on predicting the VM’s CPU and memory demands in the coming $\tau = 5$ minutes using our proposed Wiener filter with safety margin. Although our framework works for any $\tau$ value, the selection of $\tau = 5$ is based on the fact that Google traces report resource utilization for the scheduled VMs every 5 minutes. The number of the most recent observed samples considered in prediction is $L = 6$ as our experiments showed that considering more samples would increase the calculation overhead while making negligible accuracy improvements. As for the safety margin, The EWMA weight factor $\alpha$ is set to 0.25 as our experiments showed that this is the value that balanced the most between reducing the under and the over estimation error. The parameters of our migration heuristic are set to $N_{\text{on}} = 1$ and $N_{\text{sleep}} = 1$. The values of the parameters that
are specified here will be used throughout all the following experiments unless otherwise specified. For the sake of comparison, we also show in Fig. 4.3 the number of overloads that were not predicted by our predictors. Observe how low the number of unpredicted overloads is; it is actually zero with the exception of three short peaks. This proves the effectiveness of our framework vis-a-vis of predicting overloads ahead of time, thus avoiding VM performance degradation.

In order to evaluate how effective our prediction technique is compared to existing overload avoidance techniques, we report in Fig. 4.4 the total number of unpredicted overloads during the entire 24-hour trace period for the following overload avoidance techniques:
Figure 4.5: Total migration energy overhead under each of the three heuristics.

Figure 4.6: VM moving energy overhead.

Figure 4.7: The number of waken PMs to host the moved VMs.
Figure 4.8: Total migration overhead aggregated over the entire testing period (normalized w.r.t. Enhanced Largest First overheads).

Figure 4.9: Total migration overhead for the proposed migration heuristic under different values of $N_{on}$ and $N_{sleep}$ (normalized w.r.t. Lower Bound).

Figure 4.10: Number of PMs needed to host the workload.
Figure 4.11: Amount of energy that our allocation framework saves every hour when compared to allocation without overcommitment.

Figure 4.12: Google cluster’s total consumed energy during the 24-hour trace period for the different schemes.
techniques: \(i\) Fast Up Slow Down (FUSD) online prediction technique [80], and \(ii\) the Threshold-Based Avoidance Technique [68–71] where overloads are predicted when the CPU/memory utilization of the overcommitted PM exceeds 90\%. We also report in Fig. 4.4 the total number of unpredicted overloads during the entire 24-hour trace period for our proposed Wiener filter with and without safety margin. Observe from Fig. 4.4 that both the FUSD and the Threshold-based avoidance technique had a larger number of unpredicted overloads compared to our proposed Wiener filter. This shows that our proposed prediction module is more effective in terms of predicting overload compared to existing techniques. Observe also that by adding a safety margin to our proposed Wiener Prediction technique, the number of unpredicted overloads was reduced further. The gap between the case with and without safety margin quantifies the benefit of adding a safety margin to our prediction module.

**VM Migration Energy Overhead.** Having shown the efficiency of our proposed prediction module, we now evaluate how efficient our proposed migration heuristic is when compared to existing migration strategies. In order to do that, we plot in Fig. 4.5 the total migration energy overhead (including both VM moving and PM switching overheads) incurred by the migration decisions of our proposed heuristic to avoid/handle the overloads reported in Fig. 4.3 along with the total energy overhead associated with the migration decisions of the two existing heuristics:

- **Largest First [82–85]:** This heuristic considers one overloaded PM at a time, and orders all its hosted VMs in a decreasing order of their resource usage. It then starts migrating VMs from the top of the ordered list, one at a time, until the PM’s usage goes below its capacity.

- **Sandpiper [88]:** It is similar to Largest First with the exception that it uses, as a
sorting metric, the VM’s resource usage divided by the VM’s moving energy cost.

Both of these heuristics handle multi-dimensional resources by considering the product metrics, and both select the PM with the largest slack as a destination for each migrated VM.

Observe from Fig. 4.5 that our proposed heuristic incurs significantly lesser total migration overhead when compared to the other two heuristics. We next analyze the two components/sources of migration energy overhead, VM moving overhead and PM switching overhead, separately, so as to shed some light on how and what each component contributes:

1. VM Moving Energy Overhead. Fig. 4.6 shows the VM moving energy overheads under each of the three heuristics throughout the 24-hour period. Observe that our heuristic has lower moving overheads. This is due to the fact that for each overloaded PM, our heuristic identifies and migrates the VMs with the least moving cost.

2. PM Switching Energy Overhead. Fig. 4.7 shows the number of PMs that were switched ON because of VM migration. Observe that the number of PMs that forced to be turned ON when hosting migrated VMs is much smaller under our heuristic than under the other two heuristics. This is because both Largest First and Sandpiper heuristics give a higher priority to the PMs with the largest slack, but without accounting for the PMs’ power states.

The discussions and insights drawn above from Figs. 4.6 and 4.7 explain why our approach achieves lower total migration overhead when compared to the other heuristics as shown in Fig. 4.5.
Having compared the energy overhead of our migration heuristic against Largest First and Sandpiper, we now propose an enhancement for these existing heuristics so that they account for the power state of the PMs in the cluster, and evaluate our heuristic against the enhanced versions of these existing techniques. The enhancement basically makes Largest First and Sandpiper place the migrated VM on a sleeping PM only if the VM can’t be fitted in any already ON PM, whereas if multiple ON PMs have enough slack to host the moved VM, then the ON PM with the largest slack is selected for placement. This is different from the original implementation of these heuristics (as described in [82–85, 88]) where the migrated VM is placed on the PM with the largest slack regardless of whether the PM is ON or asleep.

Fig. 4.8 shows the total migration energy overhead (including both the VM migration and the PM switching overheads) aggregated over the entire 24-hour testing period for the enhanced versions of Largest First and Sandpiper as well as for our proposed migration heuristic, normalized with respect to the aggregate overheads of the Enhanced Largest First heuristic. Results show that our heuristic incurred significantly lower migration overhead than even the enhanced version of those heuristics. The main reason why these enhanced heuristics still incurred higher overheads than our heuristic is due to the fact that they both select first what VM(s) to migrate, then they search for an ON PM to fit the VM(s) selected for migration, where they don’t always succeed in finding an ON PM with enough resource slack to fit the VM(s) selected for migration and end up waking up some PMs from sleep. Furthermore, the metric that these heuristics use to combine the CPU and memory resources does not always make good VM selections for migration. Our heuristic addresses these limitations by solving a well-formulated optimization problem. Fig. 4.8 also shows the overheads of a lower bound of the optimal migration decisions, where the results show that our heuristic’s overhead is very close to
the lower bound of the optimal solution. Details on how the lower bound is calculated will be provided in the following paragraph.

**Heuristic’s Optimality and Execution Time.** We also wanted to compare the energy overhead of our heuristic’s decisions against the optimal solution obtained by solving the full optimization problem that was introduced in Section 4.5. However, the large number of integer decision variables that are involved in the full optimization problem has prevented us from finding the optimal solution within a reasonable time. This has led us to compare our heuristic against a lower bound of the optimal solution that can be calculated quickly and efficiently.

Any time a set of PMs $O_{pm}$ are predicted to overload, the lower bound for the migrations with the least total energy overhead to avoid the predicted overloads can be calculated as follows. For each PM in $O_{pm}$, the VM(s) with the least moving cost that should be removed from the considered PM to avoid the overload are identified by finding the exact solution of a two-dimensional 0-1 Knapsack problem [94]. The lower bound is then obtained by aggregating the VM moving costs of all the VMs that need to be removed from all the PMs in $O_{pm}$. This is a lower bound on the optimal solution as it identifies which VMs need to be migrated to avoid the predicted overloads such that the VM moving costs (the first part of the objective in Eq. 4.6) are minimized while completely ignoring the PM switching costs (the second part of the objective in Eq.4.6). This is the case since no CPU or memory resources are reserved for the removed VMs and thus no PM needs to be waken from sleep, and as the removed VMs are not contending over the resource slack that is available on the PMs that are ON in the cluster.\(^3\)

\(^3\)It is worth mentioning that the Knapsack-based constructed lower bound was preferred over the lower bound that can be obtained by solving the Linear Program (LP) relaxation of the optimization problem presented in Section 4.5 as our evaluations based on Google traces showed that our constructed lower bound was always tighter than the bound obtained by solving the LP relaxation. Furthermore, for the case when large number of PMs are predicted to overload, our constructed lower bound requires less
Fig. 4.9 shows the lower bound energy overhead for the migrations needed to avoid all the overloads that were predicted within the 24-hour trace period. The figure also shows the total migration energy overhead (including both VM moving and PM switching overheads) for our proposed heuristic under different values of \( N_{on} \) and \( N_{sleep} \) when handling all the predicted overloads within the 24-hour trace period. Observe that for the case of \( N_{on} = 1 \) and \( N_{sleep} = 1 \), our heuristic had a migration overhead that is around 20% larger than the lower bound of the optimal solution. Observe also that after increasing the parameters of our heuristic to \( N_{on} = 5 \) and \( N_{sleep} = 5 \), the energy overhead of our heuristic becomes less than 1% larger than the lower bound of the optimal solution. This shows that the migration decision made by our heuristic are very close to the optimal case where we know for sure that the overhead of the optimal case must be larger than or equal to the lower bound. Observe also that there is no significant difference in the total overhead of our heuristic’s migration decisions when increasing the parameters further to be \( N_{on} = 10 \) and \( N_{sleep} = 10 \).

We also show in the second column of Table 4.1 how much time on average it took our heuristic to make migration decisions any time a set of PMs were predicted to overload within the 24-hour trace period under the different parameters. The execution time is based on running a Matlab code for our heuristic on an x-86 platform that has 2 sockets, each socket has 8 cores, each core has 2 threads with a CPU frequency of 2.6 Ghz and a 62 GB RAM. The Integer Linear Program in our heuristic is solved by Matlab’s Mixed-Integer Linear Programming solver and using the default configurations of the solver as specified in [95]. Observe that on average only few seconds were needed for our heuristic to make the migration decisions needed to avoid the PMs from overloading under the different parameters. The larger the number of ON PMs \((N_{on})\) and the sleeping amount of memory resources to calculate when compared to the LP relaxation lower bound.
PMs \( (N_{\text{sleep}}) \) that were considered as candidate PMs to host the migrated VMs by our heuristic, the higher the average time needed to make the necessary migrations, but also the lower the energy overhead of the made migrations as we have seen in Fig. 4.9.

By calculating the lower bound of the optimal solution we were also able to identify some of the cases where we know for sure that the migration decisions of our heuristic were optimal. Our heuristic’s migration decisions were optimal any time a set of PMs were predicted to overload and the costs incurred by the decisions made by our heuristic to avoid the predicted overloads were exactly equal to the lower bound costs. We report in the third column of Table 4.1 how many times we know for sure that the migrations of our heuristic were optimal under the different parameter values where the reported numbers are percentages of the total times migrations needed to be performed to avoid some PMs from overloading during the entire 24-hour testing period. Observe that the percentages are very high and slightly increase as \( N_{\text{on}} \) and \( N_{\text{sleep}} \) increase as further PMs are considered as destination PMs to host the VMs selected for migration.

Table 4.1: Evaluation of the proposed migration heuristic under different values of \( N_{\text{on}} \) and \( N_{\text{sleep}} \) during the entire testing period.

<table>
<thead>
<tr>
<th>Heuristic Parameters</th>
<th>Mean Execution Time (second)</th>
<th>Percent of Times We Know that Migrations were Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (N_{\text{on}} = 1, N_{\text{sleep}} = 1) )</td>
<td>1.8</td>
<td>88.4%</td>
</tr>
<tr>
<td>( (N_{\text{on}} = 5, N_{\text{sleep}} = 5) )</td>
<td>2</td>
<td>96%</td>
</tr>
<tr>
<td>( (N_{\text{on}} = 10, N_{\text{sleep}} = 10) )</td>
<td>3.4</td>
<td>97.1%</td>
</tr>
</tbody>
</table>

**Number of Active PMs.** Since the energy consumed by ON PMs constitutes a significant amount, we analyze in Fig. 4.10 the number of ON PMs when running our framework on the Google traces under each of the three studied migration heuristics. Recall that each migration heuristic makes different decisions to handle PM overloads, and these decisions affect the number of ON PMs, as new PMs may be switched ON to accommodate the migrated VMs. We also show the number of ON PMs when no
overcommitment is applied. This represents the case when the exact amount of requested resources is allocated for each VM during its entire lifetime. By comparing these results, observe that after a couple of learning hours, our proposed prediction framework leads to smaller numbers of ON PMs when compared with the case of no overcommitment, and this is true regardless of the VM migration heuristic being used. Also, observe that our proposed prediction techniques, when coupled with our proposed VM migration heuristic, leads to the smallest number of ON PMs when compared with Largest First and Sandpiper heuristics, resulting in greater energy savings. It is worth mentioning that during the first couple of hours, the number of ON PMs is the same regardless of whether resource overcommitment is employed and regardless of the migration technique being used, simply because prediction can’t be beneficial at the early stage, as some time is needed to learn from past traces to be able to make good prediction about VMs’ future utilizations.

Energy Savings. Fig. 4.11 shows the amount of energy (in Megawatt-Hour) that Google cluster saves every hour when adapting our integrated framework (the proposed prediction approach and the proposed migration heuristic) compared to no overcommitment. Observe that savings are not substantial at the beginning as the prediction module needs some time to learn the resource demands of the hosted VMs, but these savings quickly increase over time as the predictors start to observe larger traces and tune their parameters more accurately. It is also clear from Fig 4.11 that although our framework incurs migration energy overheads (due to both VM moving and PM switching energy overheads) that would not otherwise be present when no overcommitment is applied, the amount of energy saved due to the reduction of the number of ON PMs is much higher than the amount of energy incurred due to migration energy, leading, at the end, to greater energy savings.
Finally, the total energy that Google cluster consumes when adapting our integrated framework is compared to the case when the cluster is overcommitted and uses one of the existing overload avoidance techniques, FUSD or Threshold-based, combined with each of the prior work migration heuristics, Largest First (LF) and Sandpiper (SP), to handle overloads. Fig. 4.12 shows the total cluster’s energy consumption during the entire 24-hour duration normalized with respect to the no overcommitment case. Observe from Fig. 4.12 that our framework cuts the total consumed energy by 40% compared to the no overcommitment case. Observe also that the cluster’s consumed energy when adapting our framework is significantly lower than any of the remaining overcommitment schemes. This is attributed to the fact that by predicting the resource demands of the hosted VMs accurately and by making efficient migration decisions, our integrated framework consolidates the workload using the least number of active PMs, while keeping redundant PMs in sleep state, which leads to significant lower energy consumption.

4.8 CONCLUSION AND FUTURE WORK

We propose in this chapter an integrated energy-efficient, prediction-based VM placement and migration framework for cloud resource allocation with overcommitment. We show that our proposed framework decreases the performance degradation caused by overloads while also reducing the number of PMs needed to be ON and the migration overheads, thereby making significant energy savings. All of our findings are supported by evaluations and comparative studies with existing techniques that were conducted on real traces from Google. For future work, we plan to conduct further comparative studies to evaluate our framework against other techniques and using further real traces.
Chapter 5: Workload Shifting and Energy Storage Peak Shaving

Framework

High penalties are enforced by the grid company for the peak power demand that the cloud Data Center (DC) draws within the monthly billing cycle. This chapter proposes a framework that aims at shaving the DC’s monthly peak power demand and hence reducing the DC’s monthly expenses by proposing efficient control strategies that decide: 

i) when and how much of the DC’s workload should be delayed given that the workload is made up of multiple classes where each class has a certain delay tolerance and delay cost, and

ii) when and how much energy should be charged/discharge into the DC’s batteries. We consider first the case where the DC’s power demands throughout the whole billing cycle are known and we present an optimal peak shaving control strategy. We then relax this assumption and propose an efficient control strategy when we only know predictions (accurate/noisy) of the DC’s power demands in a short duration in the future. Several comparative studies are conducted based on real traces from a Google DC in order to validate the proposed techniques.

5.1 INTRODUCTION

According to [52], large IT companies such as Google, Microsoft and Amazon spend millions of dollars per month to pay the electricity bills associated with their Data Centers (DCs). These electricity bills account for 30% to 50% of the total DCs operational
expenses [96]. Thus, there is clearly a great monetary incentive to cut down those expenses.

The electricity bill that the DC receives from the grid company at the end of the billing cycle (e.g. month) is made up of two components [97, 98]: i) Energy Charge: which is dependent on the amount of energy (measured in kilowatt hour (kWh)) the DC consumes during the entire billing cycle, and ii) Peak Charge\(^1\): which is a penalty that is proportional to the maximum amount of power (measured in kilowatts) that was drawn by the DC during the whole billing cycle. This penalty is very expensive and is enforced by the grid company to encourage the DC to balance its power demand and to discourage spiky power usage. The maximum amount of power drawn by the DC is calculated in a time-slotted fashion where the grid company calculates the DC’s average power usage during each slot of a certain length (e.g. 15-minutes), and the peak charge is calculated based on the slot with the maximum average power among all the billing cycle’s slots.

The majority of the prior techniques that were proposed to reduce the electricity bill focused exclusively on minimizing the Energy Charge while completely ignoring the Peak Charge. Unlike these techniques, this chapter focuses on minimizing the Peak Charge component of the electricity bill as it has a high contribution to the total electricity bill as will be seen in our case studies. This is achieved by shaving the peak power drawn by the DC using a controller that tunes two knobs: 1) Workload Shifting: where some of the DC’s computing jobs are delayed (and so are their corresponding power demands) during peak periods, and 2) Energy Storage: where extra power is drawn from the grid during low-demand periods and is stored in batteries so that it can be used later to reduce the amount of power drawn from the grid during peak periods.

\(^1\)Peak Charge is sometimes referred to by Demand Charge in the literature.
Three aspects make finding an efficient control strategy challenging:

- **Battery Losses and Constraints:** Batteries are not ideal devices in reality as they lose a certain percentage of their stored energy over time and these losses are called *leakage losses*. Also when routing a certain amount of energy to the battery, a certain percentage of the routed energy gets lost due to conversion operations and such losses are called *conversion losses*. Furthermore, batteries have a limit on the amount of energy that they can store and on how fast they can charge/discharge energy.

- **Workload Heterogeneity:** The DC hosts jobs with different priorities and different Service-Level Agreements (SLAs). Each class of jobs can tolerate getting its requested power demands delayed by a certain amount of time and such delays have certain delay charges.

- **Workload Uncertainty:** The DC’s workload changes over time and the duration of the billing cycle is long (typically one month). This makes it hard to make optimal control decisions as it is generally hard to know the DC’s future power demands throughout the whole billing cycle.

This chapter proposes a control strategy that considers all of the above-mentioned aspects. We start first by assuming that the DC’s power demands throughout the whole billing cycle are known in advance (*full-horizon knowledge*), and we present an approach that finds the optimal control strategy for a battery with specific losses and constraints and for a workload with different classes in terms of delay tolerance and delay charges. The proposed full-horizon approach makes optimal decision when the DC’s demands are predictable over the entire billing cycle. It also provides an upper bound of how much
monetary savings can be achieved and helps with selecting what type of battery the DC should be equipped with based on the DC’s workload demands. We then consider the case where we only know the DC’s power demand for a short duration in future (limited-horizon knowledge), where we propose an algorithm that uses this knowledge to decide at each time step how much energy the battery needs to charge/discharge and whether the demanded power should be delayed for some of the classes of jobs hosted in the DC.

In summary, our main contributions are:

- We propose peak shaving strategies that reduce DC’s electricity bill through optimal energy storage and workload shifting control decisions. We propose strategies for systems with both full- and limited-knowledge of future power demands.

- We account for real energy storage losses and constraints, and consider DCs with heterogeneous workloads, which are divided into multiple different classes, with each class having different delay tolerance and price.

Evaluations based on Google DC traces show that our control strategy achieves great monetary savings when compared to state-of-the-art techniques.

The rest is organized as follows. Section 5.2 introduces Google DC traces and provides a case study that illustrates why one should worry about reducing the DC’s Peak Charge. Section 5.3 introduces our notations and Section 5.4 describes the optimization problem that is solved in order to find the optimal full-horizon control strategy. Section 5.5 describes the limited-horizon algorithm. Section 5.6 provides comparative evaluation of our proposed approaches. Section 5.7 summarizes how our work differs from prior work. Finally, Section 5.8 concludes and provides directions for future work.
5.2 The Case for Google DC

In order to illustrate the significance contribution of the Peak Charge component to the total electricity bill, we conduct an experiment where we rely on real workload traces [47] from a Google DC that is made up of around 12K servers to calculate how much power that DC consumes over time. Google traces report the jobs that clients submitted to one of Google DCs. Each job belongs to one of four classes: non delay-sensitive, low delay-sensitive, medium delay-sensitive and high delay-sensitive\(^2\). We refer to these four classes by \(c_1, c_2, c_3\) and \(c_4\) respectively. Each submitted job is made up of a number of tasks where each task is assigned a Linux container and utilizes an amount of CPU resources over time. In order to calculate Google DC’s power consumption, we parse the traces and track at each time slot how much CPU resources are being utilized by all the tasks belonging to all the jobs that are currently hosted on the Google DC. We then calculate for each time slot what is the least number of servers needed to be kept ON to serve those tasks as if state-of-the-art Energy Charge minimizing techniques [34, 62, 92, 99–101] were applied to consolidate the DC’s workload. The power consumption of each ON server is then calculated based on the power model in [56] where the server’s consumed power, \(P_{on}\), increases linearly from \(P_{idle}\) to \(P_{peak}\) as the server’s CPU utilization, \(\nu\), increases from 0 to 100\%. More specifically, \(P_{on}(\nu) = P_{idle} + \nu(P_{peak} - P_{idle})\), where \(P_{peak} = 400\) and \(P_{idle} = 200\) Watts. The rest of the DC servers that are not hosting any tasks don’t consume any power as they are assumed to be switched off completely or put to highly power efficient sleep states to save energy. Google DC is assumed to have a Power Usage Efficiency (PUE) of 1.7, which is a typical value for DCs [102] and means that for every watt spent on IT power, an additional 0.7 watt is spent by non-computing infrastructure.

\(^2\)High delay-sensitive jobs are called production jobs in the traces and represent jobs that produce monetary revenue.
Fig. 5.1 plots the calculated power drawn by Google DC (referred to by No Peak Shaving) over the entire trace period (29 days). Fig. 5.1 also plots the power consumption of the Google DC when the same energy that was consumed by the DC during the 29-day period was spread evenly over the entire billing duration. This case is referred to by "Optimal Peak Shaving" as it represents the case where the DC had the same Energy Charge as the "No Peak Shaving" case but where the Peak Charge was minimal.

We then calculate the electricity bill for Google DC during the entire 29-day period and using real power prices [1], where the price to calculate the Energy Charge is $\alpha = 0.05\$/kWh, and the price to calculate the Peak Charge is $\beta = 20\$/kW and where the Peak Charge is calculated based on dividing the billing cycle into slots each of length $\tau = 15$ minutes. Each slot is associated with a demand power, representing the average power demanded during that slot. The peak power is then the highest among all these averages. We plot in Fig 5.2 the contribution of both the Energy Charge and the Peak

---

Google revealed workload traces for only 29 days and thus in our analysis we consider the length of the billing cycle to be 29 days rather than a month of 30 or 31 days.
Figure 5.2: Breakdown of Google DC total electricity bill (based on the power consumption in Fig. 5.1).

Charge components to the total electricity bill based on the power consumption of No Peak Shaving and Optimal Peak Shaving cases that were shown in Fig. 5.1. Observe that for the No Peak Shaving case, the Peak Charge contributes to 56% of the total electricity bill while the Energy Charge accounts for the remaining 44%. This also shows that Optimal Peak Shaving case reduces the Peak Charge paid by the DC during that month by $86K when compared with the No Peak Shaving case, which is equivalent to a 31% reduction of the total electricity bill. This translates into saving more than $1M per year. These numbers indicate how significant the contribution of the Peak Charge to the total electricity bill is, and show the potentials for reducing this electricity cost if we adopt efficient peak shaving strategies. Our proposed framework achieves this by making smart energy storage and workload shifting decisions.
5.3 Notations

We consider a time-slotted billing model, where the billing cycle is divided into $n$ slots and each slot has a duration of $\tau$ minutes. The index $i$ is used to refer to one of these slots. All the power notations mentioned next are measured in kW. To make it easier to distinguish the decision variables from the input parameters (the known quantities), small alphabetic letters are used exclusively to refer to the former whereas capital alphabetic letters or mathematical symbols (e.g. $\alpha, \eta$) are used to refer to the latter.

5.3.1 Energy Storage Notations

Fig. 5.3 illustrates the power flow during each slot $i$, where:

- $g_i$: is the power taken from the grid to serve the DC’s power demands during the $i^{th}$ slot.
- $c^+_i$: is the power taken from the grid to charge the battery during the $i^{th}$ slot.
- $t_i$: is the total power taken from the grid during the $i^{th}$ slot.
- $c^-_i$: is the power discharged from the battery to serve the DC’s power demands during the $i^{th}$ slot.
- $m_i$: is the power that our controller decides to provide to the DC in slot $i$ from both the grid and the battery. This power can be lower than the DC’s power demands in slot $i$ if the controller decides to delay providing the power demands of some of the DC’s jobs. The power can be also higher than the DC’s power demands in slot
if our controller decides to provide power for some of the jobs that were delayed from the previous time slots.

- $r_i$: is the amount of energy stored in the battery at the beginning of the $i^{th}$ slot.

5.3.2 Battery Specs

The specs of the battery are summarized by the tuple $\Phi = (\eta_c, \eta_l, C_{max}^+, C_{max}^-, R_{max})$, where:

- $\eta_c$ represents the battery’s conversion efficiency and falls within the range $(0, 1]$, and means that only $\eta_c c_i^+$ percent out of the $c_i^+$ power that the battery draws from the grid ends up being stored in the battery, whereas the remaining $(1 - \eta_c)c_i^+$ gets lost due to conversion operations.

- $\eta_l$ represents the battery’s leakage efficiency and falls within the range $(0, 1]$ and means that the battery loses $(1 - \eta_l)$ percent of its stored energy per one slot of time due to leakage losses.

- $C_{max}^+$ and $C_{max}^-$ are the maximum charging and discharging rates which represent the maximum amount of power that the battery can draw from the grid and that the battery can discharge to the data center respectively.

- $R_{max}$ represents the battery’s maximum energy storage capacity that can be used for peak shaving.

In addition to those specs that are summarized by $\Phi$, the initial energy that is stored in the battery at the beginning of the first slot in the billing cycle is referred to by $R_{init}$. 
There are three things to mention regarding the battery model. First, although the term battery is normally used to refer to a device that stores energy chemically, it is actually used in this chapter to refer to any energy-storage device regardless of the underlying storage technology as our battery model is general enough to cover the different technologies. Second, although the battery shown in Fig. 5.3 is represented as a single entity, it is actually made up of many storage cells that are normally placed in a large room in the DC’s facility [96]. Finally, the DC’s battery always stores an amount of energy called the backup energy that is enough to power the DC at full capacity for one minute while the remaining battery’s capacity is used for peak shaving. The backup energy is used to power the DC during the transition time until the Diesel generator starts providing power when a power outage occurs. According to [103], this transition time is less than a minute while the DC’s battery has usually enough capacity to power the DC at full capacity for more than half an hour.

5.3.3 Workload Shifting Notations

The DC draws power in order to execute jobs belonging to different classes. Let $C$ be the set of all the job classes that the cluster hosts. Each class $c \in C$ has a different priority and can tolerate getting the requested power demands within a Delay Window of length $T^c$ time slots from the time the power demands are requested. The longer the
length of the class’s Delay Window, the less sensitive the jobs belonging to that class are to delays. We also consider the case where the DC is charged a delay cost for not providing the requested power demands directly. For a time slot \( i \) and a job class \( c \), the following notations are used:

- \( D^c_i \): is the power demanded by class \( c \) at time step \( i \).
- \( a^c_{i,k} \): is the power requested at time step \( i \) and allocated (provided) in time step \( k \) for class \( c \) where \( i \leq k \leq i + T^c \).
- \( l^c_{i,k} \): is the power left to provide at the beginning of time slot \( k \) from the power requested at slot \( i \) for class \( c \).
- \( m^c_i \): is the power provided for class \( c \) at time step \( i \). This power could be equal to \( D^c_i \) if all the class’s requested power in slot \( i \) is directly provided with no delay and no power demands delayed from a previous slot were provided in slot \( i \). The provided power \( m^c_i \) can also be less than \( D^c_i \) if a portion of the requested power demands in slot \( i \) are delayed for a latter slot. Also \( m^c_i \) can be greater than \( D^c_i \) if some of the class’s power demands that were requested previously were deferred and were provided in slot \( i \).

5.4 **Full-Horizon Optimal Control**

Given that we know the DC’s power demands for each class \( c \in \mathbb{C} \) throughout the whole billing period referred to by \( \vec{D}^c = \{ D^c_1, D^c_2, ..., D^c_n \} \), the specs of the battery \( \Phi \), and the initial amount of energy that the battery holds \( R_{\text{init}} \), we find the optimal control strategy with the minimal electricity bill by solving the following optimization problem.
**Objective:** We seek to minimize 1) the total electricity bill which is made up of two components: Energy Charge and Peak Charge; and 2) the Delay Charge that results from not providing the requested power demands directly for the different job classes. Thus our objective can be expressed as:

\[
\text{Minimize } \alpha \xi \sum_{i=1}^{n} t_i + \beta \max_i \{t_i\} + \sum_{c \in C} \sum_{k=1}^{T_c} \sum_{i=1}^{n} \gamma_{c}^{k} a_{i,i+k}^c
\]

Where: \(\alpha\) is the energy price measured in \((\$/kWh)\), \(\xi = \tau / 60\) is a constant that converts the total energy the DC draws from the grid into kWh, \(\beta\) is the peak price measured in \((\$/kW)\), and \(\gamma_{c}^{k}\) is the cost (measured in \$/kW) for delaying the power demands of class \(c\) by \(k\) time slots.

**Constraints:** The optimization problem is solved subject to the following constraints. One,

\[
t_i = g_i + c_i^+ , \quad 1 \leq i \leq n
\]

which states that for each time slot \(i\) the total power drawn from the grid is the aggregation of the power used to sever the DC’s power demand and the power drawn to be stored in the battery. Two,

\[
g_i + c_i^- = m_i , \quad 1 \leq i \leq n
\]

which states that the power provided to the DC is taken from the grid and from the
power discharged from the battery for each time slot \(i\). Three,

\[
r_i = \begin{cases} 
R_{\text{init}}, & i = 1 \\
\eta_c c_{i-1}^+ \tau + \eta_l (r_{i-1} - c_{i-1}^- \tau), & 1 < i \leq n 
\end{cases}
\] (C.3)

Which calculates how much energy will be stored in the battery at the beginning of each slot while accounting for the conversion and leakage losses. Four,

\[
c_i^- \tau \leq r_i, \quad 1 \leq i \leq n
\] (C.4)

which states that the amount of discharged energy within the slot \(i\) that has a duration of \(\tau\) must not exceed the amount of energy that is stored in the battery. Five,

\[
c_i^- \leq C_{\text{max}}, \quad 1 \leq i \leq n
\] (C.5)

which states that the discharged power at any time slot must not exceed the maximum discharging rate that the battery supports. Six,

\[
c_i^+ \leq C_{\text{max}}^+, \quad 1 \leq i \leq n
\] (C.6)

which states that the charged power at any time slot must not exceed the maximum supported charging rate. Seven,

\[
r_i \leq R_{\text{max}}, \quad 1 \leq i \leq n
\] (C.7)

which states that the amount of energy that the battery stores is bounded by the battery’s maximum storage capacity. Eight,
\[ m_i = \sum_{c \in C} m_i^c, \quad 1 \leq i \leq n \] (C.8)

which states that the power provided to the DC at time slot \( i \) is the aggregation of the power provided for each class \( c \in C \) at that time slot. Nine,

\[ t_i, g_i, c_i^+, c_i^-, r_i, m_i \geq 0, \quad 1 \leq i \leq n \] (C.9)

which states that these decision variables are all non-negative.

Recall that each class \( c \) has a certain delay window \( T^c \) within which the requested power demands must be provided. Thus, for each class \( c \in C \) the following constraints (C.10 to C.15) must hold:

\[ m_i^c = \sum_{k = \max\{i - T^c, 1\}}^{i} a_{k,i}^c, \quad 1 \leq i \leq n \] (C.10)

Which states that the power provided for class \( c \) at time slot \( i \) is the aggregation of the power that was request at \( i \) and provided directly without delay in addition to the power deferred from previous time slots. We take \( k = \max\{i - T^c, 1\} \) as there are no power demands prior to the first time slot.

\[ l_{i,i+1}^c = D_i^c - a_{i,i}^c, \quad 1 \leq i \leq n \] (C.11)

Which states that the power left to allocate at the beginning of time slot \( i + 1 \) is equal to the amount of power that was requested and wasn’t allocated directly at time slot \( i \).

\[ l_{i,k+1}^c = l_{i,k}^c - a_{i,k}^c, \quad 1 \leq i \leq n, \ i + 1 \leq k \leq i + T^c \] (C.12)
Which states that the power left to allocate at time slot $k+1$ is equal to the amount of power that was left to allocate at the beginning of the previous time slot $k$ and that wasn’t allocated within time slot $k$.

$$l^c_{i,i+T^c+1} = 0, \quad 1 \leq i \leq n$$ (C.13)

Which prevents the power demand for class $c$ from being delayed more than the delay window $T^c$.

In order to restrict our decisions to one billing cycle, we add the following constraint which prevents deferring the power demands that are at the end of the billing cycle (which we are trying to optimize) to the following billing cycle:

$$l^c_{i,n+1} = 0, \quad n - T^c \leq i \leq n$$ (C.14)

This constraint can be relaxed if we are optimizing over multiple billing cycles. Finally,

$$m^c_i, a^c_{i,j}, l^c_{i,k} \geq 0$$ (C.15)

which states that these decisions variables are non-negative and this holds $1 \leq i \leq n$, $i \leq j \leq i + T^c$, $i + 1 \leq k \leq i + T^c$.

The formulated problem is a convex optimization problem [57] as the objective is a convex function that we seek to minimize, all equality constraint functions are affine, and all non-equality constraints are convex functions. The solution of convex problems can be found quickly and there are well-developed tools that can be used to calculate the optimal solution efficiently such as the CVX package [58], which is the one used in our implementation.
5.5 **LIMITED-HORIZON CONTROL**

We discussed previously how to find the optimal control strategy when the DC’s power demands throughout the whole billing cycle are known. We now consider the case where we only know predictions of the DC’s power demands in a short duration in the future (referred to by the Prediction Window) and we propose an algorithm that uses these predictions in order to make energy storage and workload shifting decisions at each time slot while accounting for the battery’s energy losses and for the different workload class’s delay tolerance and delay costs. A pseudo code of our proposed algorithm (Algorithm1) is presented to better illustrate our algorithm and Fig. 5.4 provides a temporal illustration of the slots that we will refer to while explaining the pseudo code of our algorithm. The pseudo code of our algorithm gets launched at the beginning of each slot \( j \) and takes the following inputs:

- \( j \) the index of the current slot for which energy storage and workload shifting decisions need to be made, where \( 1 \leq j \leq n \).
- \( \vec{D}_j \) is a vector that holds the power demanded for each class \( c \in \mathbb{C} \) at the \( j^{th} \) slot (the current slot). This vector can be expressed as: \( \vec{D}_j = \{D_{j}^{c_1}, D_{j}^{c_2}, \ldots, D_{j}^{c_{|\mathbb{C}|}}\} \)
- \( \Phi \) the specs of the battery (which were introduced in Section 5.3.B).
- \( w \) the length of the prediction window which represents the number of slots in the
future for which the DC’s power demands need to be predicted.

- \( t_{max} \) is the maximal amount of total power drawn from the grid so far up to the \( j^{th} \) slot and can be calculated as:
  \[
  t_{max} = \begin{cases} 
  0, & j = 1 \\
  \max_{1 \leq k < j} \{t_k\}, & j > 1
  \end{cases}
  \]

As illustrated in the pseudo code, for each class among the workload classes, our proposed algorithm starts first by predicting the class’s power demands in the future \( w \) slots (Line 2), where these predicted power demands are referred to by \( \hat{D}_{c,j+1}^e, \hat{D}_{c,j+2}^e, \ldots, \hat{D}_{c,j+w}^e \). These predicted demands can be obtained using any machine learning technique that provides accurate predictions. The focus of this chapter is on how to use these predictions to make energy storage and workload shifting control decisions and not on what technique to use to obtain accurate predictions. Our algorithm also fetches for each class the amount of power deferred from the previous slots (Line 3), where the notation \( L_{i,j}^c \) is used to refer to class \( c \)’s power demands that were requested at time slot \( i \) and that were not provided up to the beginning of the \( j^{th} \) slot based on our algorithm’s decisions when it was launched in the previous slots. Recall that for class \( c \) the power demands that are requested at slot \( i \) must be provided within the following \( i + T^c \) slots. Thus there are no power demands delayed from more than \( j - T^c \) previous slots for class \( c \) as our algorithm ensures that these demanded power gets allocated within their Delay Window.

Our algorithm then fetches \( r_j \) which represents the amount of available stored energy in the battery at the beginning of the \( j^{th} \) slot (basically the amount of energy left in the battery after conversion and leakage losses). Now in order to determine the best control actions, our algorithm solves in Line 6 an optimization problem called the Limited-Horizon Optimization Problem (LHOP) that is similar to the full-horizon optimization
problem (Section 5.4) but with the following key differences. First, LHOP seeks to minimize the electricity and delay costs by considering only the predicted power demands from slot $j$ to slot $j + w$ while also keeping an eye on the maximum amount of power $t_{max}$ that was drawn from the grid in the previous slots of the billing cycle. The max power demand in the previous slots is needed as the Peak Charge is calculated based on the slot with the maximum power drawn from the grid throughout the whole billing cycle. Thus LHOP basically considers the maximum power drawn from the grid from the the first slot in the billing cycle and up to the $j + w$ slots. The slots beyond $j + w$ are not considered by LHOP as it is hard to predict the DC’s power demands for more than $w$ slots. The constraints (C’.1 to C’.15) in LHOP are the equivalent of constraints (C.1 to C.15) that were explained before for the full optimization problem with the exceptions that the LHOP constraints consider only the decision variables involved in the period from slot $j$ and up to $j + w$ and that the control decisions need to meet the predicted power demands rather than the actual power demands in the $j + w$ future slot (constraint C’.11). The last key different that distinguish LHOP is that it has three additional constraints (C’16 to C’18) that are included to guarantee that the leftover power demands from the previous slots are provided within the current or future slots and before the end of their Delay Window (i.e., the power demands requested at slot $j - i$ must be provided within $j - i + T^c$ slots).

LHOP is a convex optimization problem [57] as the objective is a convex function that we seek to minimize, all equality constraint functions are affine, and all non-equality constraints are convex functions. Solving LHOP returns the best control decisions that need to be made in the period from slot $j$ and up to slot $j + w$. Our algorithm then commits only to the control decisions in the $j^{th}$ slot (Line 7) that are returned by solving LHOP, while the control decisions in each of the following slots will be determined later.
when the pseudo code of our algorithm is launched again at the beginning of each one of those following slots.

5.6 Evaluation

We rely on the real power prices and on the power demands for Google DC that were introduced in Section 5.2, and we conduct comparative experiments to evaluate how much money Google DC ends up saving in a one billing cycle when using our peak shaving control strategy. Our evaluations are organized as follows. We start first by evaluating our controller when making only energy storage decisions for peak shaving where the workload demands are not allowed to be delayed. We evaluate next our controller when making only workload shifting decisions. Finally, we evaluate our controller when making both energy storage and workload shifting decisions.

5.6.1 Peak Shaving Through Energy Storage

In this subsection, we consider the case where the DC’s workload is not allowed to be shifted (i.e., $T^c = 0 \ \forall c \in C$). We study the savings that our controller achieves when making only energy storage decisions when Google DC is supplied by each of the following energy storage technologies:

- Lead-Acid (LA): this battery uses electrochemistry to store and to discharge energy.

- Lithium-Ion (LI): relies also on electrochemistry but uses different chemical components where the cathode is a lithiated metal oxide and the anode is a graphite
Algorithm 3 LimitedHorizonControl( j, D_j, Φ, w, t_max )

1: for each c ∈ C do
   Predicted Power Demands

2: [D_{j+1}^c, D_{j+2}^c, \ldots, D_{j+w}^c] ← predictFutureDemands( w )
   Previous Leftovers

3: [L_{j-T-\tau}, \ldots, L_{j-2,\tau}, L_{j-1,\tau}] ← getPreviousLeftovers( )
   end for

4: R_j ← getAmountOfStoredEnergy()

5: Solve Limited-Horizon Optimization Problem (LHOP):

Minimize

\[
\alpha \xi \sum_{i=j}^{j+w} t_i + \beta \max \{ \max_{j\leq i\leq j+w} \{ t_i, t_{\text{max}} \} + \sum_{c\in C} \sum_{k=1}^{T_c} \sum_{i=j-T_c}^{j+w} a_{i,i+k}^c \}
\]

subject to

\( t_i = g_i + c_i^+ \) , \( j \leq i \leq j + w \) \hspace{1cm} (C'.1)
\( g_i + c_i^- = m_i \) , \( j \leq i \leq j + w \) \hspace{1cm} (C'.2)
\( r_i = \begin{cases} R_j & i = j \\ \eta_i c_{i-1}^+ r_i + \eta_i (r_{i-1} - c_{i-1}^- r_i) & j < i \leq j + w \end{cases} \) \hspace{1cm} (C'.3)
\( c_i^- \tau \leq r_i \) , \( j \leq i \leq j + w \) \hspace{1cm} (C'.4)
\( c_i^- \leq C_{\text{max}}^c \) , \( j \leq i \leq j + w \) \hspace{1cm} (C'.5)
\( c_i^+ \leq C_{\text{max}}^{\text{max}} \) , \( j \leq i \leq j + w \) \hspace{1cm} (C'.6)
\( r_i \leq R_{\text{max}} \) , \( j \leq i \leq j + w \) \hspace{1cm} (C'.7)
\( m_i = \sum_{c\in C} m_i^c \) , \( j \leq i \leq j + w \) \hspace{1cm} (C'.8)
\( t_i, g_i, c_i^+, c_i^-, r_i, m_i \geq 0 \) , \( j \leq i \leq j + w \) \hspace{1cm} (C'.9)

for each c ∈ C

\( m_i^c = \sum_{k=\max(i-T_c,1)}^{\min(i,j+1)} a_{k,i}^c \) , \( j \leq i \leq j + w \) \hspace{1cm} (C'.10)
\( l_{i,j+1}^c = \begin{cases} D_i^c - a_{i,j}^c & i = j \\ D_i^c - a_{i,j}^c & j < i \leq j + w \end{cases} \) \hspace{1cm} (C'.11)
\( l_{i,k+1}^c = l_{i,k}^c - a_{i,k}^c \) , \( j \leq i \leq j + w \) , \( i+1 \leq k \leq i + T_c \) \hspace{1cm} (C'.12)
\( l_{i,j+T_c+1}^c = 0 \) , \( j \leq i \leq j + w \) \hspace{1cm} (C'.13)
\( l_{i,n+1}^c = 0 \) , \( j + w - T_c \leq i \leq j + w \) \hspace{1cm} (C'.14)
\( m_i^c, a_{i,j}^c, t_{i,k}^c \geq 0 \) \hspace{1cm} (C'.15)
\( r_{i,j+1}^c = R_{i-j,1}^c - a_{i-j}^c \) , \( 1 \leq i \leq T_c \) \hspace{1cm} (C'.16)
\( l_{i,j+k}^c = l_{i-j,i+k-1}^c - a_{i-j,i+k-1}^c \) , \( 1 \leq i \leq T_c \) , \( 1 < k \leq T_c - i + 1 \) \hspace{1cm} (C'.17)
\( l_{i,j+k}^c = 0 \) , \( 1 \leq i \leq T_c \) \hspace{1cm} (C'.18)

end for

7: Make Control Actions Specified by \( (t_j, g_j, c_j^+, c_j^-, m_j, m_j^c) \)
carbon.

- **Ultra-Capacitors (UC):** uses a double layer electrochemistry to store energy between the electrodes.

- **Fly-Wheels (FW):** is a mechanical energy storage device that uses the momentum of a wheel/cylinder to store energy.

- **Optimal (OPT):** represents an ideal battery that has zero conversion and leakage losses and unlimited charging/discharging rate.

The first four types represent the most popular energy storage technologies that are found in DCs [1], whereas the OPT battery represents an unrealistic ideal battery. The specs of these batteries are presented in Table 5.1 and are based on [1]. We also assume that there is no stored energy initially at the beginning of the billing cycle when evaluating the different types of batteries (i.e., \( R_{\text{init}} = 0 \)). These battery specs will be used throughout the chapter unless otherwise specified.

<table>
<thead>
<tr>
<th></th>
<th>LA</th>
<th>LI</th>
<th>UC</th>
<th>FW</th>
<th>OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conversion Efficiency (%)</td>
<td>75</td>
<td>85</td>
<td>95</td>
<td>95</td>
<td>100</td>
</tr>
<tr>
<td>Leakage Efficiency (% per day)</td>
<td>99.7</td>
<td>99.9</td>
<td>80</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>Max Charging Rate (mega Watt)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>( \infty )</td>
</tr>
<tr>
<td>Max Discharging Rate (mega Watt)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>( \infty )</td>
</tr>
<tr>
<td>Max Storage Capacity (mega Watt hour)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>( \infty )</td>
</tr>
</tbody>
</table>

5.6.1.1 Full-Horizon Control Evaluation

Fig. 5.5 shows the total electricity bill for running the Google DC for a one billing cycle under different scenarios, where the total bill is broken down into the Energy Charge and
the Peak Charge for each scenario. The "No Peak Shaving" scenario represents the case when the DC’s power demands are drawn only from the grid without using any battery for peak shaving. The other scenarios in Fig. 5.5 show the total electricity bill for Google DC when different types of batteries were used to shave the peak where each type of those batteries is operating based on the decisions of the full-horizon controller that was proposed in Section 5.4. The results clearly highlight that the DC’s total electricity bill can be reduced significantly if our proposed full-horizon control technique was used to control how much energy needs to be charged/discharged over time for the different types of batteries. Observe that the total electricity bill is lower for the LI and LA battery types when compared to the FW and UC as the former types have lower leakage losses than the latter types, which allows storing larger amount of energy to be used to shave the peak that is encountered later, without leaking much of their stored energy over time. The Energy Charge of the FW, UC, LI and LA batteries are slightly higher than those of "No Peak Shaving" due to the leakage and conversion losses which increase the amount of energy that the DC consumes over time. However, these extra Energy Charge leads into significant reduction in the Peak Charge which leads in turn into significant reduction of the total electricity bill.

We vary next the energy storage capacity $R_{\text{max}}$ for each type of battery and we report in Fig. 5.6 the total electricity bill of Google DC (normalized w.r.t. No Peak Shaving total costs) when the proposed full-horizon controller is making charging/discharging decisions. As expected, for each battery type the larger the energy storage capacity, the higher the amount of power that can be shaved and the lower the total electricity bill. Observe that increasing the energy storage capacity for the FW battery causes a negligible additional reductions in the electricity bill as this energy storage technology is highly leaky. This shows that from a peak shaving perspective and for the Google
Figure 5.5: Full-horizon control monetary savings for the different types of batteries based on Google traces.

Figure 5.6: Google DC total electricity bill for different battery technologies that are operating by our proposed full-horizon controller under different energy storage capacities.
traces, an FW battery with a capacity of 4 MWH is as effective as one with a 16 MWH capacity. The former requires less facility space and has a lower capital costs and thus would be a better choice when considering the energy storage capacity of the battery that the DC should be equipped with. In fact, the monetary savings in Fig. 5.6 together with the capital expenses and the facility space limitation can be used to decide what battery technology and what energy storage capacity the DC should be equipped with.

5.6.1.2 Limited-Horizon Control Evaluation

Fig. 5.7 reports the total electricity bill associated with running Google DC for a one billing cycle when different controllers are used to make charging/discharging decisions for the different types of batteries. More specifically, for each battery type, Figure 5.7 reports the electricity for each of the following controllers:

- **No Peak Shaving**: represents the case where all the DC’s power demands are provided solely from the grid where the battery is not charging/discharging any energy.

- **Full-Horizon**: represents the controller proposed in Section 5.4 which makes the optimal control decisions for each battery type as it has full-knowledge of the DC’s demands within the whole billing cycle.

- **Limited-Horizon (Oracle)**: is the control algorithm that we proposed in Section 5.5 when operating under 100% accurate predictions of the DC’s power demand in each of the following four slots (i.e., \( w = 4 \)).

- **Limited-Horizon (Noisy)**: is similar to the previous case with the exception that a random noise drawn from a Gaussian distribution with zero mean and a
standard deviation of 200 kilo Watts is added to the predicted power demand that are provided to our algorithm in each time step. The added noise represents prediction errors and can take either a positive or negative value to mimic over or under estimation of the predicted power demands.

- **Threshold:** is a well-known technique [104,105] that compares the DC’s demanded power at each slot against a fixed threshold. If the demanded power is below the threshold, then the power difference is charged into the battery within that slot. Otherwise, the battery discharges the difference (or whatever amount of energy less than the difference that is stored in the battery). The charging (discharging) power is capped s.t. it doesn’t exceed the battery’s maximum charging (discharging) rate and s.t. no extra energy beyond the battery’s capacity is being charged. Tuning the threshold for each battery type is done by evaluating different threshold values on one-day traces where the value that yielded the least electricity expenses is picked for evaluation.

Observe from Fig. 5.7 that for each type of battery the Limited-Horizon control (both Oracle and Nosy cases) had a lower total electricity bill than the No Peak Shaving case and than the Threshold controller. As expected, the Noisy case had higher electricity bill than the Oracle case due to the added prediction errors that affected slightly the decisions of our algorithm. Obviously the total electricity bill for the Limited-Horizon control is higher than that of the Full-Horizon control as the latter has the advantage of knowing the DC’s power demand throughout the whole billing cycle which allows it to make the optimal battery control decisions. Notice that the gap between the Full-Horizon and the Limited-Horizon controller is small for FW when compared to the remaining battery types. This is attributed to the fact that FW is highly leaky (has low leakage
efficiency and hence high leakage losses), thus the amount of energy that FW stores decays quickly over time which leaves only a small energy that can be used for peak shaving in the future slots that are far in time. Thus for highly leaky batteries (such as FW), knowing the future power demands for only a short duration in the future for the Limited-Horizon controller makes reductions in the electricity bill that are close to the Full-Horizon (optimal) case. These results are when the Limited-Horizon Algorithm relied only on predictions of the DC’s power demands in the following four slots. Recall that each slot has a duration of 15 minutes and thus this represents the case where our algorithm knows the predicted power demands in a short duration of one hour in the future.

We increased next the length of the Prediction Window and plotted in Fig. 5.8 the total electricity bill for Google DC (normalized w.r.t. No Peak Shaving total electricity costs) when the Limited-Horizon (Oracle) controller is making charging and discharging decisions for each battery type. The results clearly show that for each battery type, the larger the length of the Prediction Window, the higher the peak power that can be shaved,
and hence the lower the total electricity bill. For a certain length of Prediction Window, the gap between the OPT battery and any other battery technology is attributed to the leakage and conversion losses (as the OPT battery does not have any leakage or conversion losses). It is worth mentioning that for the FW battery, increasing the length of the Prediction Window leads into very small additional reductions as this battery is highly leaky and thus only a small percentage of the drawn energy at any time slot will remain in the battery to use in the slots that are far in time. It is also worth mentioning that for all types of batteries, in the extreme case when the length of the Prediction Window of the Limited-Horizon (Oracle) controller is equal to the length of the billing cycle, then the Limited-Horizon (controller) performs exactly as the Full-Horizon controller.

We investigated next in Fig. 5.9 how much time it took the Limited-Horizon controller to solve the optimization problem and to make charging/discharging decisions under different Prediction Window lengths. Since an optimization problem needs to be solved at the beginning of each time slot in order to make the appropriate power decisions,
we track for each slot how much time it took to solve the optimization problem and then we show bar plots for these execution time under each Prediction Window. The measured times are calculated based on running the Matlab code of our controller on a machine that has a CPU frequency of 2.6 Ghz and a 62 GB RAM. The variability for each Prediction Window in Fig. 5.9 is attributed to the fact that the inputs (e.g. the power demands) vary from an optimization problem to another which affect the time needed to find the optimal solution. Observe that increasing the length of the Prediction Window does increase the time needed to make control decisions as the optimization problem has now larger number of variables that need to be found. However, it took always very small amount of time (less than a second) to solve the optimization problem in all of those cases which is something vital for online DC management.

In order to illustrate the decisions that our controllers make, we plot in Fig. 5.10 the power demands of Google DC over time and we plot in a different color the total grid power (the aggregation of the power provided to the DC and the power charged into the battery) for the Full-Horizon and for the Limited-Horizon (Oracle) controllers.
when operating an LA battery. The power demanded by the DC at any point in time is provided by our controllers. This means that each time the total grid power for any of those controllers goes above the DC’s demands then the difference is being routed to be stored in the battery. Also each time the total grid power is below the DC’s demands then the difference is drawn from the battery to supply the DC’s demands. Fig. 5.10 clearly shows that the DC’s peak power demand at the 21\textsuperscript{st} day was shaved significantly by each of our controllers. This reduced the Peak Charge and hence minimized the DC’s electricity bill.

5.6.2 Peak Shaving Through Workload Shifting

Recall that the jobs that are reported in Google traces are classified into four classes: $c_1$, $c_2$, $c_3$ and $c_4$ where the class with a lower index number contains the jobs that are less delay-sensitive. In this subsection, we consider the case where the DC is not supplied by a battery that can store energy for peak shaving purposes (i.e., $R_{\text{max}} = 0$) but where the
power demands of some of the jobs belonging to certain classes are allowed to be delayed for a certain duration. Throughout the chapter, the costs for delaying the demanded power is made linearly proportional to the number of slots for which the requested power demands were delayed for each class. More specifically, $\gamma^c_k = k\lambda^c$ where $\gamma^c_k$ is the cost for delaying 1 KW of the power demands of class $c$ for $k$ time slots and $\lambda^c$ is the cost for delaying 1 KW of the power demands of class $c$ for one time slot. We evaluate next both the Full-Horizon and the Limited-Horizon controllers when making workload shifting decisions.

5.6.2.1 Full-Horizon Control Evaluation

We consider first the case where only the power demands of the jobs belonging to classes $c_1$ and $c_2$ (the least delay-sensitive jobs) can be delayed to be provided within a Delay Window of length $T^{c_1}$ slots and $T^{c_2}$ slots respectively and with costs $\lambda^{c_1} = 0.01$ $$/KW$ and $\lambda^{c_2} = 0.02$ $$/KW$ respectively. In this experiment, we consider the case where: $T^{c_1} = T^{c_2}$ (i.e., both classes have the same Delay Window length). We then increase the length of the Delay Window for those two classes and we plot in Fig. 5.11 the total expenses of Google DC (normalized w.r.t. "No Peak Shaving" expenses) when our proposed Full-Horizon controller is making workload shifting decisions. For each case, the total expenses are broken down into the Energy Charge, the Peak Charge and the Delay Charge. Observe that for the case when the Delay Window has a zero length, there is no Delay Charge as our controller is basically forced to provide the requested power demands directly with no delay which resulted in total expenses that are the same as the "No Peak Shaving" case. Observe that as the length of the Delay Window increases, the Peak Charge decreases and so does the DC’s total expenses as our controller gains larger
ability to delay the requested power demands over longer periods, which increases the amount of power that can be shaved. Notice that the Energy Charge for the different cases is the same as in all those cases the energy consumption (in KWH) was identical as our controller provides all the requested power demands (either directly or after some delay).

5.6.2.2 Limited-Horizon Control Evaluation

We consider the case where classes $c_1$ and $c_2$ are allowed to be delayed but when each one of those classes has a different Delay Window length where $T^{c_1} = 3$ slots and $T^{c_2} = 1$ slot. We show in Fig. 5.12 Google DC’s total expenses (normalized w.r.t. No Peak Shaving) and broken down into the Energy, Peak and Delay charge components for the Full-Horizon (FH) controller as well as for the Limited-Horizon (Oracle) and the Limited-Horizon (Noisy) controllers when knowing predictions of the future power demands in the following hour. Observe that the LH (Oracle) controller had a total
expenses that are only 3% larger than the FH controller. This shows that knowing the future power demands for a short duration of one hour in the future achieves very close reductions to those of knowing the power demands throughout the whole billing cycle. As expected, the LH (Noisy) controller had slightly higher total expenses than the LH (Oracle) controller due to the prediction errors which affected the decisions made by this controller. Fig. 5.13 further illustrates how the Peak Charge was reduced by the Full-Horizon and the Limited-Horizon (Oracle) controller where we show how the power demands for the four classes were allocated over time for the "No Peak Shaving" case and for the Full-Horizon and the Limited-Horizon (Oracle) controllers. Observe that the Full-Horizon and Limited-Horizon controllers avoided the peak at the 21\textsuperscript{st} day by delaying the power demands of the Non delay-sensitive and low delay-sensitive classes and spreading them over the following slots. This reduced the Peak Charge and lead into significant reductions in the the DC's total expenses.
Figure 5.13: Breakdown of Google power consumption for the different job classes.
5.6.3 Peak Shaving Through Energy Storage & Workload Shifting

We now evaluate our control strategies when making both Energy Storage and Workload Shifting peak shaving decisions.

5.6.3.1 Full-Horizon Control Evaluation

We evaluate first in Fig. 5.14 our Full-Horizon controller where we compare Google DC’s total expenses when making only Workload Shifting decisions (referred to by WS), when making only Energy Storage decisions (referred to by ES), and when making both Energy Storage and Workload Shifting decisions (referred to by ES+WS). The results clearly show that our controller makes the most reductions in the total expenses when making both Energy Storage and Workload Shifting decisions as our controller stores energy during the low power periods (the valleys) that are before the peak power demand and then delays some of the power demands within the peak periods to a latter time that has a low demand. Observe that the Delay Charge for the (ES+WS) case is lower than the (WS) as using the battery to store energy for peak shaving purposes reduces the time needed to delay the workload in order to shave the peak which in turn reduces the Delay Charge.

5.6.3.2 Limited-Horizon Control Evaluation

In our final experiment, we evaluate in Fig. 5.15 our Limited Horizon controller when making both Energy Storage and Workload Shifting decisions (referred to by ES+WS), when making only Energy Storage (ES), and when making only Workload Shifting (WS)
decisions. The results clearly show that by controlling the two knobs: storing energy and workload shifting, our controller was able to achieve more reductions in the expenses than the remaining cases.

The results in Fig. 5.14 and Fig. 5.15 are when the DC is supplied by an LA battery that has the following specs: $C_{max}^+ = C_{max}^- = 8MW$, $R_{max} = 8MWH$ and leakage and conversion efficiency as specified in Table 5.1. Only the non delay-sensitive jobs (class $c_1$) and the low delay-sensitive jobs (class $c_2$) were allowed to be delayed within these experiments where $T^{c_1} = 3$ slots, $T^{c_2} = 1$ slot, $\lambda^{c_1} = 0.02 $$/KW and $\lambda^{c_2} = 0.04 $$/KW.

The evaluations in Fig. 5.15 for the Limited-Horizon controller are when knowing one day ahead predictions of the future power demands.

### 5.7 Related Work

This sections summarizes the key difference that distinguish our proposed framework from prior work.
Figure 5.15: Google DC’s total expenses when our Limited-Horizon controller is making only Workload Shifting, only Energy Storage and both Energy Storage & Workload Shifting decisions. Results are normalized with respect to “No Peak Shaving” costs.

5.7.1 Energy Charge Minimization Techniques

Researchers suggested techniques such as energy-aware scheduling [34, 92, 106], Virtual Machine (VM) migration within the DC [62, 99] and server over-booking [100, 101] in order to minimize the DC’s electricity bill. All of these techniques reduce the Energy Charge component of the electricity bill by consolidating the DC’s workload on fewer number of ON servers which allows switching a larger number of servers to sleep to save energy. The main limitation of these techniques [34, 62, 92, 99–101, 106] is that they completely ignore the Peak Charge which accounts to more than 56% of the monthly electricity bill as was illustrated in the Google case study (Section 5.2). Our work complements these techniques and differs from them in that it minimizes the DC’s electricity bill by reducing the Peak Charge through energy storage and workload shifting control.
5.7.2 Energy Storage Peak Shaving Techniques

Reinforcement learning was used in [107] to make battery charging/discharging decisions based only on the DC’s current power demand and the amount of stored energy. The main limitation of this technique is that it requires tuning some parameters empirically such as the learning rate and the discretization level. Furthermore, it discretizes the input and output spaces into a finite set which leads into less accurate decisions. Markov Decision Process (MDP) was used in [108] to derive a battery control strategy for storing energy during low price periods so that it can be used later during high-price periods. Unlike our work, this technique does not target minimizing the Peak Charge which is calculated based on the maximum power consumption within the billing cycle. The Threshold technique [104] tries to operate the DC at a fixed power by charging the difference between a fixed threshold and the DC’s power demands into the battery or by discharging from the battery the difference between the DC’s power demand and the threshold. Bounds on the performance of this technique were derived in [105] using the concept of arrival curves from Network Calculus. Although the Threshold approach is simple, it is highly sensitive to the selected threshold. This was demonstrated in our evaluations (Section 5.6) where we showed that our control strategies achieve higher monetary savings than this technique. The authors in [109, 110] applied the Threshold technique on multiple hierarchical levels of a DC that has a distributed UPS topology where each server is attached to an independent battery. The focus of the framework proposed in this chapter is on a DC with a centralized UPS topology rather than the distributed topology considered in [109, 110]. Dynamic Programming (DP) was proposed in [111] to find the optimal control strategy for a battery with no losses under full-knowledge of the future power demands. Unlike the the DP approach, our proposed full-
horizon controller solves a well-formulated convex optimization problem that considers the battery’s losses and power constraints. Finally, a key distinction of our controller from all the techniques mentioned in this subsection is that it considers workload shifting as an additional control knob to achieve further peak shavings.

5.7.3 Workload Modulation Peak Shaving Techniques

The Peak Charge was reduced in [112, 113] by dropping some of the DC’s requested or scheduled jobs within periods with high power demands. Clients in these cases clearly experience service disruptions as their dropped jobs are required to be resubmitted and to start again from scratch. The work in [114, 115] explored redirecting/migrating jobs from the DC with high peak power demands into other DCs that have lower demands in order to reduce the Peak Charge. The authors in [116] capped the power consumed by the DC’s servers in addition to performing inter DC migrations. The migration approaches [114–116] are complementary to our work as they can be applied on top of our controller to balance the workload among multiple DCs. The work in [117] considered delaying and dropping jobs for peak shaving. Our proposed framework differs from [117] in that we consider the case where the DC’s workload is divided into multiple classes where each class has a different delay tolerance and a different delay cost. Another key difference from [117] is that we control two knobs simultaneously workload shifting and energy storage. To the best of our knowledge, our proposed peak shaving controller is the first that adjusts both of these knobs.
5.8 Conclusion and Future Work

This chapter proposes peak shaving strategies that aim at minimizing the DC's electricity bill by making smart energy storage and workload shifting decisions. Our proposed control strategies are based on solving a well-formulated convex optimization problem that takes into account real battery's specs such as leakage and conversion losses, maximum charging/discharging rate and storage capacity. The formulated optimization problem also considers the case where the DC's workload is made up of jobs belonging to different classes where each class has a different delay tolerance and delay charge. We propose first an optimal control strategy that operates when knowing the DC's power demand throughout the whole billing cycle. We then develop an algorithm that operates when knowing only predictions (accurate/noisy) of the DC's power demands in a short duration in the future. Several comparative studies were conducted based on real traces from a Google DC where results show that our controllers achieve promising reductions in the DC's monthly electricity bill. For future work, we plan to develop accurate workload prediction techniques and to consider other factors in our control strategy such as the battery's lifetime. We also plan to investigate operating a battery that has a distributed topology rather than a centralized one.
Chapter 6: Dissertation Conclusions

We presented in this thesis four complementary frameworks that aim at reducing the energy consumption and the electricity bill of cloud data centers.

The framework proposed in Chapter 2 predicts the future VM requests that the cloud data center will receive in future and keeps only the right amount of servers needed to server the workload ON while switching the remaining servers to deep sleep states to save energy. Predicting the cloud workload is achieved by dividing the workload into multiple categories using clustering techniques then estimating for each category the number of requests that should be received in future using a trained adaptive Wiener filter. Evaluations based on Google traces show that our framework achieves 30% reductions in the consumed energy when compared to the case where all the servers in the data center are kept ON all the time. The energy consumption of our framework is only 5% more than the optimal oracle case which assumes knowing exactly both the future VM requests that will be received in future in addition to the amount of requested computing resources associated with each future request.

In Chapter 3, we presented a resource allocation and scheduling frameworks that exploits the elasticity and the varying charging costs among the submitted requests and that decides where to place the heterogeneous submitted task requests, and how much resources should be allocated to the elastic ones such that the cloud profits are maximized while meeting all tasks demands. Experiments show that our framework achieves great reductions in the data center’s expenses and up to 25% profit increase when compared to schedulers that don’t exploit the elasticity nor the price heterogeneity among the
submitted tasks.

In Chapter 4, we presented an overcommitment framework that decides the amount of resources that should be overbooked in the data center’s servers and that makes smart VM migrations decisions in order to avoid overloading the overbooked servers. These migrations decisions are done with two objectives in mind: minimizing the number of ON servers and minimizing the migration energy overheads. Experiments show that our frameworks reduces the energy consumption of Google cluster by 40% when compared to the case when the cloud resources are not overbooked. Our framework energy reductions are also around 20% more than state-of-the-art existing overbooking techniques.

Finally, in Chapter 5 we proposed a peak shaving framework that minimizes the cloud data center’s electricity bill by making smart workload shifting and energy storage decisions. Our framework considers the case when the workload is divided into multiple classes where each class has a different delay tolerance and a different delay cost. Our framework also accounts for both conversion and leakage losses that are associated with storing energy in data center’s batteries. Experiments show that our framework can achieve significant reductions in the data center’s monthly electricity bill.
Bibliography


[103] R. Urgaonkar, B. Urgaonkar, M. Neely, and A. Sivasubramaniam, “Optimal power cost management using stored energy in data centers,” in *ACM SIGMETRICS*


