PRIORITY SCHEDULING FOR HETEROGENEOUS WORKLOADS IN COMPUTING CLUSTERS

by

Derya Çavdar

B.S, in Computer Engineering, Boğaziçi University, 2007

M.S, in Computer Engineering, Boğaziçi University, 2009

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering

Boğaziçi University

2015

PRIORITY SCHEDULING FOR HETEROGENEOUS WORKLOADS IN COMPUTING CLUSTERS

APPROVED BY:

Prof. Fatih Alagöz (Thesis Supervisor)	.....
Prof. Özgür Barış Akan	.....
Prof. Mehmet Ufuk Çağlayan	.....
Assist. Prof. Özlem Durmaz İncel	.....
Prof. Tuna Tuğcu	.....

DATE OF APPROVAL: 03.12.2015

ACKNOWLEDGEMENTS

I am grateful to my thesis advisor Fatih Alagöz for his invaluable contributions to this thesis and for his great understanding. I would like to thank thesis committee members Özgür Barış Akan, Mehmet Ufuk Çağlayan, Cem Ersoy, Özlem Durmaz İncel, Tuna Tuğcu for their feedbacks and contributions to this work. I also would like to thank to Lydia Y. Chen. I learned a lot from her and I had such an amazing time at IBM Research Zurich.

My special thanks go to my friends at CMPE. I always enjoyed working in such a friendly environment. I would like to thank to all my friends for the wonderful time I had.

I would like to express my deepest gratitude to my mother, father and brother. I am always thrilled and feel cheerful with their unconditional love and support throughout my life. During the long journey of this thesis study, I am feeling lucky to have such wonderful parents. This thesis is dedicated to them. Lastly, my heartfelt thanks must go to Haşim Sak. He was always there for me to bring bliss in my life.

During this thesis, I was supported by the State Planning Organization of Turkey (DPT) under the TAM Project with Grant No. DPT-2007K 120610 and Turk Telekom Research Center.

ABSTRACT

PRIORITY SCHEDULING FOR HETEROGENEOUS WORKLOADS IN COMPUTING CLUSTERS

Nowadays, large computing clusters constantly strive for an optimal tradeoff between resource efficiency and performance. In this thesis, we are concerned with the efficient use of system resources and we also aim to improve response time of the tasks. We tackle with the challenges of task scheduling on large heterogeneous clusters executing highly heterogeneous bursty workloads with different priorities, resource demands, and performance objectives. Firstly, we propose a scheduling algorithm for tasks with communication needs which improves the response time and resource utilization by controlling communication and computation resources at the same time. Secondly, we propose a novel scheduling framework for exploring various aspects of priority scheduling with heterogeneous workloads while investigating the tradeoff between evictions and response time. To better understand the impact of evictions, we first analyze simple eviction policies and wasted resources associated with evictions by using trace-driven simulations. Furthermore, by exploiting the heterogeneity of the workload, we propose a workload-aware slot configuration and task assignment methodology incorporated with slot-based priority scheduling to improve class-based response time and resource efficiency. Finally, we introduce a task scheduling policy which aims to provide scheduling and execution guarantees for low priorities while preserving the performance benefits of high priority tasks. The proposed scheduling effectively handles both prioritization and performance issues of low priorities by utilizing a combination of preemptive and non-preemptive scheduling.

ÖZET

HETEROJEN İŞ YÜKLERİ İÇİN ÖNCELİKLİ ÇİZELGELEYİCİ TASARIMI

Günümüzde veri merkezleri, kaynak verimliliği ve performans arasında optimal bir dengeye ulaşmaya çalışır. Bu tezde, sistem kaynaklarının verimli kullanımı ile ilgilenilmiş ve iş bitirme süresinin iyileştirilmesi amaçlanmıştır. Çok değişken iş yüklerinin, heterojen büyük veri merkezlerinde çizelgelenmesinde yaşanan zorluklar ele alınmıştır. İlk olarak iletişim gereksinimleri olan işler için iletişim ve hesaplama gereksinimlerini birlikte kontrol ederek kaynak verimliğini ve iş bitirme sürelerini iyileştiren bir çizelgeleme algoritması önerilmiştir. İkinci olarak, öncelikli çizelgelemenin çeşitli yönlerini keşfetmek ve iş tahliyeleri ile iş bitirme süresi arasındaki ödünleşimi analiz etmek için yeni bir çizelgeleme çatısı önerilmiştir. İş tahliyelerinin sistem performansı üzerindeki etkilerini daha iyi anlayabilmek için, temel tahliye yöntemleri önerilmiş ve gerçek iz kullanılarak performans değerlendirmesi yapılmıştır. Bunun yanında, bu tezde iş yükleri heterojenliği kullanılarak, öncelikli çizelgeleme ve iş yükü farkındalığı olan bir çizelgeleme algoritması önerilmiştir. Önerdiğimiz yöntem ile iş çizelgelemesine iş yükü farkındalığı entegre edilerek, iş bitirme süresi ve kaynak verimliliğinde önemli iyileştirmeler elde edilmiş ve başarım çalışmaları ile gösterilmiştir. Son olarak, düşük öncelikli işler için çalıştırılma garantisi sağlanırken aynı zamanda yüksek öncelikli işlerin performansını da gözeten yeni bir çizelgeleme planı önerilmiştir. Önerilen yöntem hem önceliklendirme hem düşük öncelikli işlerin performans sorunlarını hibrit bir çizelgeleme yaklaşımıyla başarıyla ele almaktadır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES x
LIST OF TABLES xiv
LIST OF SYMBOLS xvi
LIST OF ACRONYMS/ABBREVIATIONS xx
1. INTRODUCTION 1
1.1. Key Contributions 2
1.2. Organization of the Thesis 3
2. RELATED WORK 5
2.1. Scheduling in Large Computing Clusters 5
2.2. Workload Analysis 6
2.3. System Design 7
2.4. Other approaches 8
3. NETWORK AWARE TASK SCHEDULING 10
3.1. System Model 10
3.1.1. Data Center Topology 10
3.1.2. Cost Model 12
3.1.2.1. Servers 12
3.1.2.2. Network Elements 14
3.2. Problem Formulation 15
3.2.1. Complexity of the MTEC Problem 19
3.3. Proposed Algorithms 20
3.3.1. Best Energy Path Scheduling (BEPS) 20
3.3.2. Best Resource Fit Scheduling (BRFS) 20
3.3.3. Round Robin Scheduling (RRS) 21
3.4. Performance Evaluation 22
3.5. Chapter Summary 25

4. SCHEDULING FOR WORKLOADS WITH COMMUNICATION NEEDS . 26
4.1. Motivation . . . 26
4.2. System Model . . . 29
4.2.1. The MapReduce Workloads . . . 29
4.2.2. Network Requirement . . . 30
4.2.3. Data Center Architecture . . . 31
4.2.4. Energy Consumption . . . 32
4.2.4.1. Server Energy Consumption . . . 32
4.2.4.2. Network Energy Consumption . . . 33
4.2.5. Response Time Model . . . 33
4.3. Green MapReduce Scheduler . . . 34
4.3.1. GEMS Phase I . . . 34
4.3.2. GEMS Phase II . . . 37
4.4. Performance Evaluation . . . 37
4.4.1. Comparing with noSleep Policy . . . 38
4.4.2. Comparing with noNetwork Policy . . . 39
4.4.3. Scalability Analysis . . . 41
4.5. Chapter Summary . . . 42
5. RESOURCE INEFFICIENCIES OF LARGE CLUSTER SCHEDULERS . . 43
5.1. Google Cluster Trace . . . 43
5.1.1. Workload Heterogeneity . . . 44
5.2. Analysis of Eviction . . . 45
5.2.1. Cause of Eviction . . . 45
5.2.1.1. Machine Failure . . . 46
5.2.1.2. Task Resource Overcommitment . . . 47
5.2.1.3. Preemption by Higher Priority . . . 47
5.2.2. Eviction per Priority . . . 48
5.2.3. Workload Analysis of Priorities . . . 49
5.3. Chapter Summary . . . 50
6. A FLEXIBLE, HOLISTIC SCHEDULING FRAMEWORK FOR LARGE CLUSTERS . . . 53
6.1. Motivation . . . 53

6.1.1. Impact of Evictions 54
6.1.2. Impact of Overbooking 55
6.1.3. Contributions 56
6.2. System Model 57
6.2.1. Task Model 58
6.2.2. Server Model 59
6.2.3. Scheduler Model 59
6.2.3.1. Slot-based Resource Assignment 59
6.2.3.2. Priority Scheduling 61
6.2.3.3. Priority Eviction Model 62
6.2.3.4. Memory Exceed 63
6.2.4. Response Time Model 65
6.2.4.1. Successful Execution 65
6.2.4.2. Waiting Time 67
6.2.4.3. Wasted Time 67
6.2.5. Dynamic Power Management 67
6.2.6. Energy Consumption Model 68
6.3. Workload and Server Environment 69
6.3.1. Server Heterogeneity 69
6.4. Design Comparisons 70
6.4.1. Performance Metrics 70
6.4.2. Effect of Priority Eviction Policy 72
6.4.3. Effect of Memory Eviction Policy 73
6.4.4. Effect of Priority and Memory Eviction Policies 75
6.5. Chapter Summary 76
7. PRIORITY SCHEDULING FOR HETEROGENEOUS WORKLOADS . . . 77
7.1. Motivation 77
7.2. System Model 79
7.3. Priority Scheduling Analysis 79
7.3.1. Preemptive Priority Scheduling 80
7.3.2. Impact of Priority Evictions 81
7.3.3. Understanding the Reasons under Repetitive Evictions 84

7.3.3.1. Simulation Verification 88
7.3.4. Limiting Eviction 89
7.4. Chapter Summary 93
8. WORKLOAD-AWARE PRIORITY SCHEDULING 94
8.1. Workload-aware Slot Configuration and Task Assignment 95
8.1.1. Workload-aware Slot Configuration 95
8.1.1.1. Phase I 96
8.1.1.2. Phase II 98
8.1.2. Workload-aware Scheduling Policy 99
8.1.3. Complexity of WA^{2}SC 100
8.2. Performance Evaluation 101
8.2.1. On Evictions: Priority Evictions vs. Memory Evictions 101
8.2.2. On System Utilization 103
8.2.3. On Response Time 105
8.2.3.1. Impact of Class Arrivals on R 105
8.2.3.2. Improvement on Class-based Response Time 106
8.3. Chapter Summary 108
9. HYBRID SCHEDULING POLICY FOR PRIORITY-BASED OPERATION OF COMPUTING CLUSTERS 109
9.1. Motivation 109
9.2. HYBRID Scheduling Policy 111
9.2.1. Phase-I 112
9.2.2. Phase-II 113
9.3. Performance Evaluation 114
9.3.1. Impact on Evictions 115
9.3.2. Impact on Class-based Response Time 116
9.3.3. Impact of the Number of Sticky Slots 118
9.3.4. Evaluation of HYBRID Scheduling with Diverse Workloads . . . 119
9.4. Chapter Summary 122
10. CONCLUSIONS 124
10.1. Summary of Contributions 124
REFERENCES 128

LIST OF FIGURES

Figure 3.1. Data center system architecture. 11

Figure 3.2. Activity profile of 5,000 Google servers over 6 months [1]. 12

Figure 3.3. Best Energy Path Scheduling (BEPS) 21

Figure 3.4. Best Resource Fit Scheduling (BFRS) 22

Figure 3.5. Energy consumption of BEPS vs. optimal and RRS. 23

Figure 3.6. Total energy consumption, network energy consumption and resource utilizations with BFRS and BEPS and savings with BEPS executing on a heterogeneous cluster of 1250 nodes. 24

Figure 4.1. GEMS (GrEen MapReduce Scheduler). 35

Figure 4.2. GEMS savings compared against GEMS (noSleep) and GEMS (noNetwork): executing MapReduce on a heterogeneous cluster of 1250 nodes. 40

Figure 4.3. Number of *on* servers executing MapReduce on a heterogeneous cluster of 1250 nodes. 41

Figure 5.1. The task state transition diagram of Google Scheduler [2]. 44

Figure 5.2. Task arrivals of different priority classes from Google Trace. 49

Figure 6.1. Average resource reservations versus actual usage in Google cluster trace (moving hourly) [3]. Production, middle and batch classes from bottom to top. 56

Figure 6.2. Main options of the simulator framework. 58

Figure 6.3. Priority scheduler design. 61

Figure 6.4. Memory eviction model: X denotes the check point to be compared against the estimated minimum execution time of the task, T_x^{min}. . 64

Figure 6.5. The breakdown structure of response time of a task. 66

Figure 6.6. The state transitions of a server with IDL and SLP power management policies. 68

Figure 6.7. Effect of priority eviction policies on WE, V and E with unlimited memory. 73

Figure 6.8. Effect of memory evictions with no-priority scheduling on WE, V and E. 74

Figure 6.9. Effect of priority eviction and memory eviction policies on WE, V and E. 75

Figure 7.1. Distribution of number of evictions of evicted tasks with MRS, RND and LRS under resume and non-resume scheme. 83

Figure 7.2. Number of evicted and non-evicted tasks of class 0 running at the time of eviction and the evicted task with MRS, RND and LRS, where ne is the number of evicted and nne is the number of non-evicted tasks at the time of eviction. 88

Figure 7.3. Number of evictions, drops, and wasted executions by priority class with MRS, RND, LRS and MRS5, RND5, LRS5. 90

Figure 7.4. Slot utilization, queue length and number of *ON* servers with MRS, RND, LRS and MRS5, RND5, LRS5. 91

Figure 7.5. Response time by priority class with MRS, RND, LRS and MRS5, RND5, LRS5. 92

Figure 8.1. Workload-aware Slot Configuration (WASC) 96

Figure 8.2. Workload-aware Scheduling Policy (WASP) 100

Figure 8.3. Number of priority and memory evictions for WA^{2}SC and workload oblivious policies SC1, SC2, SC3, and SC4. 102

Figure 8.4. Number of available slots and CPU, memory utilizations, energy consumption for WA^{2}SC and workload oblivious policies SC1, SC2, SC3, and SC4. 104

Figure 8.5. Response time by priority class, V, R. 106

Figure 8.6. Improvement on class-based response time with WA^{2}SC compared to workload-oblivious policies SC3 and SC4. 107

Figure 9.1. The system design with HYBRID scheduling. The system is configured according to workload-aware slot configuration (WASC) as shown in Algorithm 8.1. Phase II employs workload-aware Scheduling Policy (WASP) which is presented in Algorithm 8.2. 112

Figure 9.2. Impact of repetitive evictions on the response time of evicted tasks of priority class 0: HYBRID vs. WASP. 116

Figure 9.3. Improvement on class-based response time with HYBRID scheduling policy compared to WASP. 117

Figure 9.4. Class-based response time of HYBRID scheduling with 0, 200, 1536 and 3072 sticky slots. 118

Figure 9.5. Arrival patterns of 5 different trace with 60K task arrivals in 15 hours. 120

LIST OF TABLES

Table 3.1. Server configurations and power consumption of four types of servers from Google Trace. 11

Table 3.2. Data center topology: number of switches, servers, links and their power consumption. 14

Table 3.3. Summary of notations. 16

Table 4.1. Topology: number of switches and their power consumption. . . . 31

Table 4.2. Performance evaluation results of 4 hours. 38

Table 4.3. Performance evaluation results of 24 hours. 39

Table 5.1. Causes of eviction and their percentages. 46

Table 5.2. Percentage of tasks, percentage of evictions, interarrival time, resource demand and resource usage breakdown by priority. 52

Table 6.1. Design decisions of the scheduler model. 60

Table 6.2. System design decision options. 71

Table 7.1. Server configurations. 79

Table 7.2. WE and R with MRS, RND and LRS eviction policies under resume and non-resume systems. 82

Table 7.3. Analysis of eviction behavior for MRS, RND and LRS under resume and non-resume systems. 85

Table 8.1. CPU-based versus workload-aware system configurations. 98

Table 9.1. Evictions, response time and WE analysis of HYBRID and WASP. 115

Table 9.2. Trace characteristics: real, semi-synthetic and synthetic traces. . . 119

Table 9.3. Improvement on class-based response time with HYBRID [%] compared to WASP. 121

Table 9.4. Parameters used to generate semi-synthetic and synthetic traces. . 123

LIST OF SYMBOLS

b	Fitting value
c	Confidence interval indicator
C	CPU capacity of the most powerful server
C_l	CPU capacity of the server type l
c_j^l	Current capacity of resource l in server j
$b(u, v)$	Current bandwidth capacity of link (u, v)
d	Number of resource dimensions in a server
E	Energy consumption
$\overline{e}$	Average number of evictions per evicted task
$E(\gamma)$	Energy cost of path γ
$E_{network}$	Energy consumption of network
E_{server}	Energy consumption of servers
E_{total}	Total energy consumption
$G(N, L)$	Graph representation of the data center topology
H	Set of all servers
$J_i(u, v)$	Binary decision variable indicating if task i uses link (u, v)
K	Set of priority classes
L	Set of edges which represents links
$LRSl$	Least Recently Started policy with dropping threshold l
$MRSl$	Most Recently Started policy with dropping threshold l
$max(e)$	Maximum number of evictions a task experiences
$max(pe)$	Maximum number of priority evictions a task experiences
N	Set of vertices which represents nodes consisting of switches and servers
N_c	Number of cores
ne	Number of evicted tasks at an instant
ne^*	Denotes that the evicted task is from already evicted tasks
NE	Total number of evictions
$NE_{priority}$	Number of priority evictions

NET	Total number of evicted tasks
NET_n	Number of evicted tasks with policy n
N^j	Set of nodes connected to a server j
nne	Number of non-evicted at an instant
nne^*	Denotes that the evicted task is from non-evicted tasks
np	Non-production class
$[N_p N_{np}]$	Slot configuration matrix of (non-)production tasks
$N_r(t)$	Number of running tasks at time t
N_s	Number of slots
N_u	Set of nodes connected to a switch u
N_w	Number of tasks waiting to be scheduled
Opt_{mtec}	Optimal solution of MTEC problem
Opt_{msec}	Optimal solution of MSEC problem
p	Production classes
P	Number of paths in the architecture
$\mathbf{p}$	Best path
P_{act}	Activation cost
P_{cpu}	Power consumption of CPU
P_i	Percentage of arrivals from priority class i
P_{idle}	Power consumption of a server while on *idle* mode
$P_{idle \rightarrow on}$	Power cost of transitioning from *idle* to *on*
P_{mem}	Power consumption of memory
$P_{on \rightarrow idle}$	Power cost of transitioning from *on* to *idle*
$P_{on \rightarrow sleep}$	Power cost of transitioning from *on* to *sleep*
P_{op}	Operating cost of a server
P_{peak}	Maximum power consumption of a server
P_{server}	Power cost of a server
P_{sleep}	Power consumption of a server while on sleep mode
$P_{sleep \rightarrow on}$	Power cost of transitioning from *sleep* to *on*
$P_{server}(j)$	Power cost of server j
$P_{switch}(u)$	Power cost of switch u

$\overline{pe}$	Average number of priority evictions per evicted task
q	Best fit server
R	Response time
r	Root node
$R(C_i)$	Average response time of class i
$R(C_i)_{evicted}$	Average response time of evicted tasks of class i
$R(C_i)_{non-evicted}$	Average response time of non-evicted tasks of class i
$R(C_i, pe = j)$	Average response time of tasks of class i that experience j priority evictions
$R(C_i, pe > j)$	Average response time of tasks of class i that experience more than j priority evictions
$RNDl$	Random policy with dropping threshold l
s	Sink node
S	Set of switches
$\mathcal{S}_k$	Experimental setting k
t	Time slot
$T_{idle \rightarrow on}$	Delay cost of transitioning from *idle* to *on*
t_{KI}	Scheduling time of the evictor task
t_{KO}	Time stamp of eviction
$T_{on \rightarrow idle}$	Delay cost of transitioning from *on* to *idle*
$T_{on \rightarrow sleep}$	Delay cost of transitioning from *on* to *sleep*
T_q	Time spent on queueing
$T_{response}$	Response time of a task including execution and waiting time
T_x	Time spent for successful execution
T_x^{min}	Estimated minimum execution time
$T_{sleep \rightarrow on}$	Delay cost of transitioning from *sleep* to *on*
T_w	Wasted time of a task
T_{wait}	Waiting time of a task including queueing and wake up time
T_{we}	Time spent on wasted executions
$T_{we}(P)$	Time spent on wasted executions due to priority
$T_{we}(M)$	Time spent on wasted executions due to memory exceed
T_{wu}	Time spent on waking up a server

U_{cpu}	Normalized CPU utilization
U_{mem}	Normalized memory utilization
V	Weighted response time metric
w	Weight vector
W_j	Binary decision variable indicating whether server j is *on*
WE	Wasted executions
X	Time of check pointing
$X_{u,v}$	Binary decision variable indicating whether link (u, v) is *on*
Y_u	Binary decision variable indicating whether switch u is *on*
α_i	Computational requirement of task i
α_i^c	CPU requirement of task i
α_i^m	Memory requirement of task i
α_i^l	Resource requirement l of task i
β	Percentage of CPU resource of a server
γ'	Minimum energy path for external communication
$\Gamma_c(t)$	CPU rate
δ	Slow down sensitivity factor
Δ_C	CPU demand of a task
η	Selected task for eviction
θ_i	Estimated CPU usage of priority class i
μ_i	Bandwidth requirement of task i
ξ_i	Mean CPU usage of priority class i
ρ	Wake up time of a server or switch
σ_i	Standard deviation of CPU usage of priority class i
τ	Time threshold
ϕ_i	Percentage of CPU occupancy of priority class i
φ_i	CPU occupancy of priority class i
ψ'	Minimum energy path for internal communication
ω_i	Mean CPU demand of priority class i

LIST OF ACRONYMS/ABBREVIATIONS

APX	Approximable
BEPS	Best Energy Path Scheduling
BFRS	Best Resource Fit Scheduling
CDF	Cumulative Distribution Function
DC	Data Center
DFS	Dynamic Frequency Scaling
DPM	Dynamic Power Management
DVFS	Dynamic Voltage Frequency Scaling
FCFS	First Come First Serve
GEMS	GrEen MapReduce Scheduler
HYBRID	Hybrid Scheduling Policy
ICT	Information and Communication Technology
IDL	Remain Idle
LPF	Lowest Priority First
LRS	Least Recently Started
LSF	Last Started First
MRS	Most Recently Started
MSEC	Minimization of the Server Energy Consumption
MTEC	Minimization of the Total Energy Consumption
PUE	Power Usage Effectiveness
QoS	Quality of Service
RND	Randomly Selected
RRS	Round Robin Scheduling
SC	Slot Configuration based on CPU Capacity
SLO	Service Level Objective
SLP	Immediate Sleep
VM	Virtual Machine
WAA	Workload-Aware Assignment
WAE	Workload-Aware Eviction

WASC	Workload-Aware Slot Configuration
WASP	Workload-Aware Scheduling Policy
WA^{2}SC	Workload-Aware Slot Configuration and Task Assignment

1. INTRODUCTION

Large computing clusters have become the standard IT delivery platform whose efficiency is one of the foremost optimization criteria in daily operations. Executing heterogeneous workloads with different priorities, resource demands and performance objectives is one of the key operations for today's data centers to increase resource efficiency. On the other side, due to the ever increasing complexity of systems and dynamicity of workloads [3], scheduling policies employ techniques, e.g., duplicative executions, to mitigate the impact of unexpected events and further avoid the degradation of tail response times [4]. As a result, computation resources are unavoidably spent on such executions which are mainly caused by priority evictions [5]. While a vast number of studies provide the "green" evidence of data centers, such as PUE, little is known about how the computing resources are given away without resulting in useful computation.

In particular, the major challenge for task scheduling in data centers is to provide efficient resource management among different applications with diversified requirements. The complexity of schedulers is exacerbated by burstiness and required performance objectives of the workloads. Therefore, in this thesis we address the following questions: (i) how a scheduler should make task assignment in a heterogeneous server environment with multi dimensional resources attaining both response time and resource efficiency, (ii) how to minimize the impact of task evictions on system performance and resource efficiency, (iii) how to incorporate the workload arrival patterns, resource demands and resource usages of task classes in scheduling decision,

In this thesis, first we propose to control task assignments in combination with the sleeping policy of servers and switches. Next, we focus on priority scheduling where tasks belong to classes with different resource requirements. Providing class specific response times on heterogeneous clusters is highly challenging due to high variability in the resource demand of classes, i.e., some tasks take much longer time to complete or require more resources than other tasks. Our objective is to increase resource efficiency

as well as response time of task classes. Moreover, we are concerned with the degrading impact of unsuccessful executions due to evictions.

1.1. Key Contributions

Main contributions of this thesis can be listed as follows:

(i) Scheduling of workloads with communication needs: As one of the key workloads of today's data centers, MapReduce workloads – parallel execution of small tasks belonging to a job – requires communication in addition to computational resources. This type of workloads alternates between computation and communication intensive phases. In that respect, we propose a task scheduling algorithm for this type of workloads that considers CPU, memory, and external/internal network requirements of tasks and provide a response time model considering the impact of executing multiple tasks. We formulate the problem and investigate optimal solution in Chapter 3 and we introduce GEMS algorithm in Chapter 4 for this purpose.

(ii) Resource inefficiencies in large computing clusters: In order to better understand the brown side of task scheduling and further mitigate resource inefficiency in scheduling, we focus on a particular scheduling event "task eviction" which is triggered by the scheduler. The first step of our study is to qualitatively identify its dominant root cause by analyzing the sequence of tasks co-executed on the same server. Based on real cluster trace characterization analysis, we pinpoint the relationship between priority scheduling and task eviction that leads to resource inefficiencies, i.e, wasted resources in computing clusters in Chapter 5.

(iii) Priority scheduling for heterogeneous workloads: We develop a comprehensive scheduling framework which enables us to explore large design space of priority scheduling and further mitigate resource inefficiency. We introduce several eviction policies showcasing the fact that evictions can greatly degrade the response time and system performance in Chapter 6 and in Chapter 7. We quantify the wasted resources and performance degradation due to evictions with different policies and demonstrate that certain eviction policies lead to repetitive evictions

which results in outlier response times. Furthermore, we investigate and verify the reasons of repetitive eviction and we propose a policy to impose eviction threshold.

(iv) Workload-aware task scheduling: The workload heterogeneity is a key factor to be considered in order to optimize task scheduling. Hence, we propose a task scheduling algorithm which includes workload-awareness on both slot configuration and task assignment since workload oblivious approaches lead to non-optimal decisions due to incompatible resource requirements and allocations. We introduce a workload-aware algorithm in Chapter 8 which incorporates the arrival patterns, resource demands and resource usages of task classes in scheduling decision.

(v) Hybrid scheduling: The long occupation of cluster resources with long-running tasks of high priority classes may lead to starvation of low priority classes. Therefore, we propose a new scheduling policy called HYBRID that can resolve such a pitfall while preserving the performance benefit of high priority task in Chapter 9.

In short, in this thesis we are focused on the problem of task scheduling in large heterogeneous computing clusters. We are concerned with the task response time and we also aim to improve resource efficiency of the system.

1.2. Organization of the Thesis

The presentation of this thesis is organized as follows. First, in Chapter 2 we review the related works in order to clearly identify our main contributions in the literature. In Chapter 3, we present the optimization of joint server and network resource allocation problem. In Chapter 4, we propose a task scheduling algorithm for workloads with communication requirements. In Chapter 5, we investigate and analyze the main reasons of resource inefficiency from workload characterization of a real production trace. According to our findings in Chapter 5, we design and implement a flexible, comprehensive task scheduling framework to analyze different scheduling and eviction policies in Chapter 6. In Chapter 7, by using the proposed framework, we investigate the effects of priority scheduling, evictions and eviction policies on system performance for highly heterogeneous workloads. We also investigate the reasons for

repetitive task evictions and propose policies to avoid outlier response times. Chapter 8 presents a novel workload-aware scheduling framework which attains better response time and resource utilization, aiming to better satisfy the varying resource requirements of the workload. In Chapter 9, we propose a new scheduling policy which utilizes workload-aware framework and aims to provide execution guarantees and mitigate outlier response times. Finally, Chapter 10 concludes this thesis by summarizing our main contributions.

2. RELATED WORK

In this chapter, we provide the related background on efficient resource and workload management in today's computing environments. Lastly, we provide other approaches to achieve performance and resource efficiency improvements.

2.1. Scheduling in Large Computing Clusters

Wide variety of tasks executed in modern data centers introduces priority scheduling in order to meet the application specific performance objectives [2, 5–7]. Despite ignoring task priorities, some studies achieve significant power savings by focusing on workload consolidation and dynamic right-sizing such as [8–10] and [11–13]. Some others focusing on resource scheduling with a central theme of fairness [14–16]. Related simulation work focusing on task scheduling includes [6] whose goal is to minimize scheduling delay. The related proposals deal with aggregate workloads, hence the allocations are done based on overall resource demand, overlooking the task level constraints, requirements and performance objectives. As a consequence, these designs overlook priorities and evictions.

There are some priority schedulers proposed for Hadoop system [17, 18] which is mainly based on providing scheduling precedence for high priorities. Other designs which focus on priority scheduling mainly work with two priority classes only in order to simplify the model [19–21], as opposed to multi-class priority scheduling that we consider in this work. Typically, existing approaches work with simplified models, i.e., at most two priority classes [22, 23] with non-preemptive or resume systems [24] since non-resume schemes introduce instabilities [25]. Despite the fact that priority scheduling is a common approach to provide guaranteed services in today's systems [2, 26, 27], the analytical studies relies on low number of classes and work conserving environments. A recent survey of analytical and simulation approaches focusing on cloud solutions in general is presented in [28].

The problem of designing eviction policies with class-based response time minimization objective has received very little attention although evictions are one of the major causes of inefficiency in large clusters. There exist eviction strategies for MapReduce workloads which rely on the exact knowledge of task duration or remaining time, but this information is not usually available. [5] focuses on job and task eviction policies with hard deadlines while global preemption [29] is seeking for fairness on job and task level. Amoeba [30] adopts preemption to meet performance levels for a work conserving environment with the overhead of checkpointing. Unlike our study, these papers do not provide a comprehensive comparison of eviction policies for resume and non-resume systems.

Our work is differentiated from this literature by generality of the model considered which includes comprehensive response time and eviction models and proposes a priority scheduling framework which captures wasted resources and provides workload-aware solutions.

2.2. Workload Analysis

Understanding of workload properties and patterns through workload analysis is one of the major aspects for developing new scheduling techniques to achieve resource efficiency and better response time. The focus of the related trace analysis is on providing an overall view of statistical properties about all scheduling events and resource utilizations of the system resources [3, 31, 32]. The Google cluster trace [33] – which we use in our work – is extensively characterized by numerous studies focusing on task classification, task and job dependency and heterogeneity and dynamicity of the workload [31, 34–37]. The trace characterization studies not only provides a better understanding of the workload complexity but also reveals insights for designing new efficient schedulers. Even though characterization of heterogeneous traces is well investigated, quantitative analysis of wasted resources due to unsuccessful events has not gain attention.

Motivated by the significant number of unsuccessful events in the trace, we turn

our attention to causes of failures and their impact on system performance and resources. There is a number of studies focusing on software and hardware failures mainly on distributed systems. Related studies on failure analysis extends storage [38,39] and network units [40–43], operating systems and computing infrastructure [44–46]. These studies mainly provides the causes, types and frequency of failures. However, the dependency between unsuccessful events and system performance mainly response time and quantification of resource efficiency in terms of useful computation remains unstudied. Hence, we fill this gap by providing field trace analysis on the quantification of the impact of unsuccessful events on system performance.

The model introduced in this thesis shows a holistic approach that optimizes priority task scheduling by employing workload-awareness. There is a large body of literature on workload characterization, mainly focusing on system analysis [3,34,47], rather than the impact of unsuccessful events. Differentiated from the characterization studies in the literature, we address and quantify the resource inefficiency in large clusters as a result of heterogeneity via field trace analysis with Google Cluster trace. We further extend this line of work by concentrating our efforts on mitigating the non negligible impact of evictions. An important difference between these models and this study is apart from analyzing different eviction policies and system settings, this study provides integration of workload-awareness to the system and further improvement on class-based response time and resource efficiency.

2.3. System Design

Interest in resource efficiency [48–50] has been growing. In spite of the fact that data centers execute mixed applications with diverse requirements, the workload heterogeneity is usually overlooked and turns into a tough challenge for designing schedulers and systems. Several trace characterization studies [34,51,52] employ real cluster trace data from Google, Yahoo! and Facebook. Most remarkable characteristic observed is heterogeneity: in terms of tasks (priority, execution time, CPU/RAM requirement) and server environment (cores, CPU, memory). The level of heterogeneity observed from real trace characterization studies explains the scale of the complexity

that the system faces.

We adopt slot-based resource assignment in our system, where slot only limits the number of running tasks on a server, unlike Hadoop slots [53], slots are not fixed size nor hold dedicated resources. In Hadoop systems, a node is configured with a fixed number of fixed size type slots, i.e., map and reduce slots, where the number of slots is estimated by some heuristics irrespective of workload characteristics. Since the number of slots in a Hadoop cluster is fixed throughout the lifetime of the cluster, most of the proposed solutions can be reduced to a variant of slot type decision (map or reduce) problem. Although some recent research has proposed dynamic slot configuration [54, 55], the dynamic adjustment stays in the slot type level, i.e, converting map to reduce or reduce to map slot according to the system state. Other studies has focused on resource-aware scheduling for Hadoop to alleviate the inefficiency of fixed slot configuration [56, 57] at the job scheduling level. Recently, the next generation Hadoop scheduler has been introduced as a resource model consisting of a container which is like our slot description [27]. Our proposal differentiates from Hadoop system by deciding number of slots based on workload. This approach introduces heterogeneous slot configuration, in which some servers are equipped with less number of slots hence more powerful slots to accommodate high resource demanding high priority tasks.

2.4. Other approaches

Apart from the approaches listed above, there are other approaches to indirectly improve resource efficiency or system performance. Virtualization technology utilizes the consolidation of different applications on the same server by transforming applications to be platform independent. Platform independency allows virtual machines (VMs) to be migrated to different servers when necessary [58]. Virtualization also benefits from energy savings by reducing active number of servers via consolidation. Recently, virtualization tools are provided by some vendors like VMware [59] and Xen [60].

Several papers addressed the optimization of VM placement [61] and migra-

tion [62] for cloud computing environments. EnaCloud [63] focused on live application placement for cloud platforms. The proposed approach uses VMs to place applications and achieves energy savings by allowing live migrations to most power efficient servers when possible. In GreenCloud project, Buyya et al. [64] study on energy efficient resource allocation and scheduling algorithms with QoS consideration. The proposed mixing and mappings of VM approach is evaluated by simulation using the CloudSim [65]. Although virtualization allows to consolidate different applications thereby obtaining substantial power savings, it is hard to optimize which workloads to colocate in order to minimize interference between applications [66, 67].

There are other proposals in the literature that address performance issues in large clusters. Systems providing parallel executions of small tasks employs *speculative executions* in order to meet tail latency SLOs [68–71]. In particular, the underlying system needs to determine the straggler tasks to initiate speculative executions. Therefore, the controller needs to wait and check progress of tasks which incurs a significant overhead. Additionally, speculative executions increase the system load. In order to reduce the wasted executions, *checkpointing* [30, 31] is widely studied. However, the system overhead of checkpointing is really high and it is hard to find the safe points to checkpoint the task state. Another approach to improve the performance of tasks is *blacklisting* [3, 72]. Blacklisting relies on identifying the slow machines mainly due to hardware failures and avoiding scheduling tasks to blacklisted machines. However, it is difficult to identify machines to be blacklisted due to complex nature of the problems.

3. NETWORK AWARE TASK SCHEDULING

The increasing demand for computation, storage and network has lead to deployment of large data centers. Efficient management of multidimensional resources is a challenging task with changing workload arrivals in terms of resource utilization with a possible energy cost. In this chapter, we approach resource management problem of large data centers from the potential savings that can be attained via putting idle servers and network elements in sleep mode, while satisfying the computation and communication requirements of the incoming tasks. First, we formulate the problem as an optimization problem and show that it is not in APX, i.e., there is no polynomial time constant-factor approximation scheme. Therefore, we propose two scheduling algorithms which take into account the cost of both data center networking and computing elements for scheduling and routing decisions for the tasks. Our simulation experiments show that our proposed scheduling algorithms achieves substantial savings by allowing sleep mode for idle servers and network elements.

3.1. System Model

In this section, we present the heterogeneous data center environment that we used in our analysis and different operation/power states of servers as well as switches.

3.1.1. Data Center Topology

In our system model, we consider a 3-tier data center architecture as shown in Figure 3.1. The first layer is composed of core switches which are the source nodes where the incoming traffic enters the data center. The nodes in the next two levels are aggregation and access switches, respectively. The last node layer is heterogeneous servers which have different CPU, and memory capacities. Node layers are connected through links, which have different bandwidth capacities according to the connected layers.

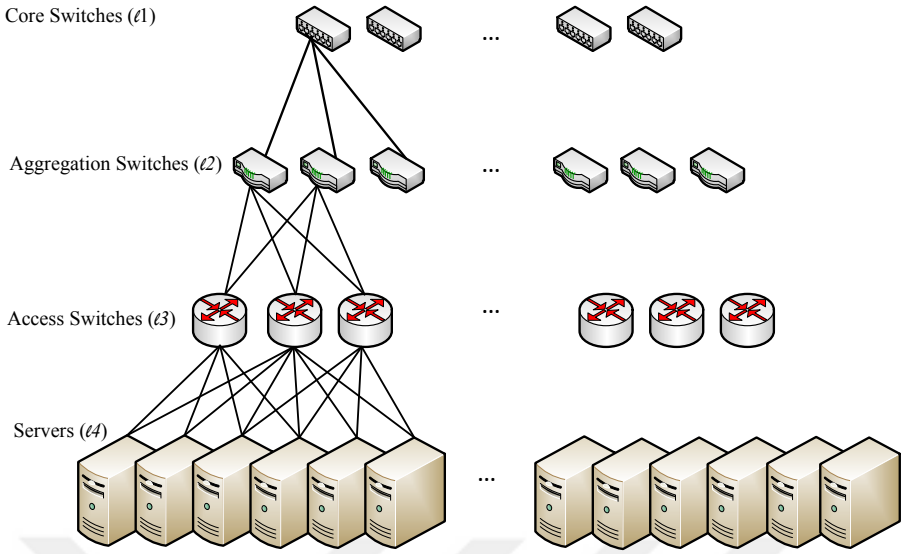


Figure 3.1. Data center system architecture.

By using Google trace data in [73], we consider four types of server nodes whose respective percentage and CPU/memory capacities are listed in Table 3.1. Google trace also provides data on tasks' CPU, memory requirements and their corresponding execution time. All task requirements and machine resource capacities are normalized with a range between [0,1] according to the machine with maximum capacity. However, power consumption values of servers and network requirement of tasks are not provided.

Table 3.1. Server configurations and power consumption of four types of servers from Google Trace.

		Normalized		Power [watts]				
	Ratio	CPU	Memory	P_{cpu}	P_{mem}	P_{sleep}	P_{idle}	P_{peak}
A	54%	0.5	0.5	51.5	18	46.25	162	231.25
B	31%	0.5	0.25	51.5	9	44.5	162	222.5
C	8%	0.5	0.75	51.5	24	47.5	162	237.5
D	7%	1.0	1.0	103	36	60.2	162	301

In our study, we use two different DCs. The large DC is composed of 1250 servers while the small DC is composed of 125 servers, as shown on Table 3.2. For the small DC, we use 120 minutes time span. On the other hand, we use 24 hours for the large

DC to include hourly task arrival fluctuations in our analysis.

3.1.2. Cost Model

Because of the strict quality of service (QoS) concerns, data centers are designed to handle peak load. In addition to the QoS requirements, data centers should be fault tolerant and highly available. These requirements make data centers to be high energy consuming environments. Despite the need for high availability of data centers, it is shown that servers are idle most of the time [1]. In that regard, idle servers and the connected network elements are consuming significant amount of energy. In other words, leaving server in idle mode wastes a lot of energy. Therefore, there is an opportunity to achieve energy efficiency for data centers by adopting energy proportionality. Energy proportional designs allow devices to consume power according to the activity level, i.e., not consuming power when idle. In practice, it is hard to achieve ideal energy proportionality. As an approximation, a good strategy is to design computing and networking devices with multiple sleep states having different power consumption levels.

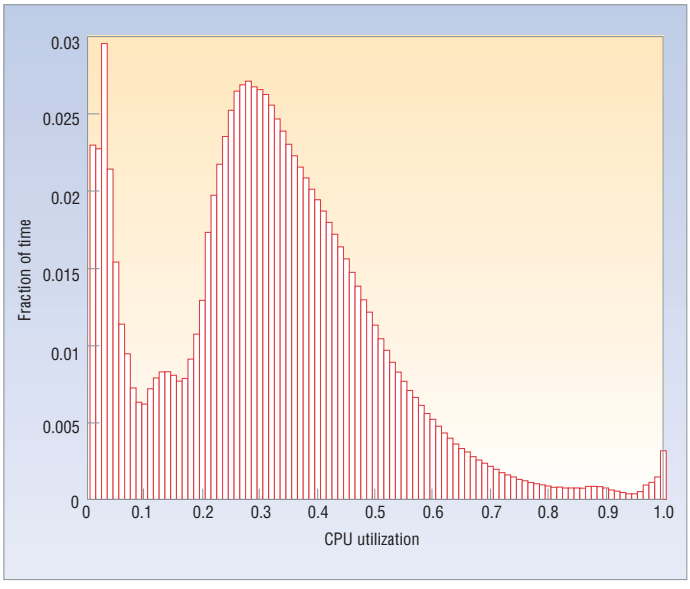


Figure 3.2. Activity profile of 5,000 Google servers over 6 months [1].

3.1.2.1. Servers. There exist two major solutions to achieve energy savings in computing servers: dynamic voltage frequency scaling (DVFS) [74] and dynamic power

management (DPM) [75]. The DVFS scheme adjusts voltage frequency hence CPU power consumption according to the system load. As the frequency is reduced, the execution is slowed down but energy savings are obtained in terms of CPU power. However, DVFS is only effective on CPU while the other components still run at unreduced power levels.

In contrast to DVFS, DPM scheme can save more energy by powering down all the components of computing servers. DPM allows dynamically adjusting the number of in-/active servers while satisfying the QoS requirements of the system. *Idle* server consumes around 60% of the *on* server while fully utilized since *idle* servers keep main components active in order to be responsive to the load [10]. However, it is known that servers are busy 30% of the time on average [1] as shown in Figure 3.2. Energy savings can be obtained from putting idle devices to sleep mode when they are not actively used [75]. Servers consume less in *sleep* mode than in *idle* mode, and the wake up cost of transitioning from *sleep* to *on* mode is significantly lower than transitioning from *off* to *on* in terms of time and energy cost.

Power consumption of a typical server is composed of two terms: fixed term and load-dependent term. The load-dependent term in power consumption of a server increases proportional to the memory and CPU usage. The fixed term, i.e., power consumption in idle state, covers the remaining part of the energy consumed in a server. The load dependent component for an *idle* server is zero, but fixed term still applies. We model the cost of a server by *(i)* operating cost which depends on CPU utilization and memory capacity, *(ii)* activation cost in terms of energy which is incurred due to turning on a sleeping server.

In our system, a server can be put into sleep mode if no task is being processed on that server. However, switching a server from *sleep* state to *on* state incurs energy cost called *activation cost*. Activation cost is the energy consumed during wake up of server hence calculated as the product of wake up time (ρ) and peak power consumption of the server as: $P_{act} = P_{peak}\,\rho$. The transition from *idle* to *sleep* mode incurs zero energy cost. If the server is in sleep mode than P_{server} is the sum of operating cost and

activation cost. However, if the server is in on state, P_{server} is equal to operating cost P_{op}, i.e., $P_{op} = P_{idle} + P_{cpu}U_{cpu} + P_{mem}U_{mem}$, where U_{cpu} and U_{mem} are the CPU and memory utilization respectively.

Table 3.2. Data center topology: number of switches, servers, links and their power consumption.

Network Elements	quantity in Small DC	quantity in Large DC	Power	Links	Link Capacity
Core (l_1)	2	2	25W	($l_1 - l_2$)	10GE
Aggregation (l_2)	4	16	51W	($l_2 - l_3$)	1GE
Access (l_3)	52	512	75W	($l_3 - l_4$)	1GE
Servers (l_4)	125	1250	301W(max)		

3.1.2.2. Network Elements. In the network segment, we consider the energy consumption of core, aggregation, access switches and links. In this study, we consider a data center network environment, in which each type of switch include fixed number of ports and line cards. Power consumption of a switch is basically correlated with the number of ports and the number of line cards included. Since we consider different types of links according to their capacity, we include power consumption of links to the power consumption of the switches which they are attached. Table 3.2 represents the power consumption of switch types. We use the power consumption of switches used in [76] as a reference. If the server is in running state P_{switch} is equal to operating cost. However, transition from *sleep* mode to *on* mode incurs extra energy consumption. We model the cost of a network element by *(i)* activation cost in terms of energy which is incurred to turning on a sleeping network element: $P_{act} = P_{switch}\,\rho$, *(ii)* operating cost: $P_{op} = P_{switch}$.

3.2. Problem Formulation

In this work, we consider the data center model in Section 3.1 and formulate an optimal routing and scheduling problem that minimizes the total energy consumption of servers, and network elements while ensuring all tasks are scheduled at the same time. We focus on three decisions in a data center at the beginning of a time slot: (i) determining the number of active servers and switches, (ii) determining the server which processes task j, (iii) determining the routing for task j to the server.

We use $G(N, L)$ to represent the data center network where vertices are servers and switches while edges are the links. tasks are represented as $task_1, task_2, ..., task_k$, and each task is defined by $task_i = (source, destination, \mu_i, \alpha_i)$. For $task_i$, μ_i is the bandwidth requirement and α_i is the computational requirement for $i = 1, .., k$. Computational requirement for $task_i$ is composed of CPU (α_i^c) and memory (α_i^m) requirements $\alpha_i = (\alpha_i^c, \alpha_i^m)$. Since, we do not have any knowledge of task server assignments, we need to add a dummy node s to our graph G. All servers are connected to s and the destination of all tasks are s. Due to same reasons, we add a dummy root node r to $G(N, L)$. All core switches are connected to r, so all tasks are routed from r to s.

In this section, we model and formulate Minimization of the Total Energy Consumption (MTEC) problem. The objective is to minimize the total network energy consumption, and server energy consumption while adopting sleep schedules for both servers and network elements. In this formulation, resources are on links (bandwidth) and servers (CPU and memory) while the energy consuming elements are switches and servers. As its name suggests, MTEC aims to reduce the total energy consumed by servers and network elements.

Objective function of MTEC: Minimize the total network energy consumption and server energy consumption.

Table 3.3. Summary of notations.

Symbol	Description
$G(N, L)$	graph representation of DC
L	set of edges which represents links
N	set of vertices which consists of switches and servers
S	set of switches
H	set of all servers
N_u	set of nodes connected to a switch u
N^j	set of nodes connected to a server j
$X_{u,v}$	binary decision variable indicating whether link (u, v) is *on*
Y_u	binary decision variable indicating whether switch u is *on*
W_j	binary decision variable indicating whether server j is *on*
$J_i(u, v)$	binary decision variable indicating if $task_i$ uses link (u, v)
c_j^l	current capacity of resource l in server j
$b(u, v)$	current bandwidth capacity of link (u,v)
μ_i	bandwidth requirement of $task_i$
$P_{server}(j)$	power cost for server j
$P_{switch}(u)$	power cost for switch u
α_i^l	resource requirement l of $task_i$
d	number of resource dimensions in a server
r	dummy root (all core switches are connected to r)
s	dummy sink (all servers are connected to s)
ρ	wake up time for a server or switch

$$\min \left(\sum_{u \in N} Y_u P_{switch}(u) + \sum_{j \in H} W_j P_{server}(j) \right) \tag{3.1}$$

Capacity Constraints: First there are two fundamental constraints: The total

number of tasks routed along each link must not exceed the capacity limit. Second, the total number of tasks assigned to each server must not exceed the server capacity in every dimension.

$$\sum_{i=1}^{k} J_i(u,v)\mu_i \leq b(u,v)X_{u,v},\ \forall (u,v) \in L \tag{3.2}$$

$$\sum_{i=1}^{k}\sum_{u\in S} \alpha_i^l J_i(u,j) \leq c_j^l W_j,\ l = 1,\ldots,d,\ \forall j \tag{3.3}$$

Network traffic flow conservation: tasks should not be lost or created in the intermediate nodes.

$$\sum_{u\in N} J_i(u,w)\mu_i - \sum_{y\in N_w} J_i(w,y)\mu_i = \tag{3.4}$$
$$\begin{cases} -\ \mu_i & \text{if } w = r \\ \mu_i & \text{if } w = s \quad \forall w \in N, \forall i \\ 0 & \text{otherwise} \end{cases}$$

Demand satisfaction: Each source and sink sends or receives an amount equal to its demand since we have a stable system.

$$\sum_{w\in N} J_i(r,w)\mu_i = \sum_{w\in N} J_i(w,s)\mu_i = \mu_i,\ \forall i \tag{3.5}$$

Dependency between link and switch decision variable: When a switch u

is in *sleep* mode, all links connected to this switch are also in *sleep* mode. Also when all links connecting to a switch are sleeping, the switch can enter *sleep* mode.

$$X_{u,w} = X_{w,u} \leq Y_u, \forall u \in N,\ \forall w \in N_u \tag{3.6}$$

$$Y_u \leq \sum_{w \in N_u} X_{w,u},\ \forall u \in N \tag{3.7}$$

$$X_{j,w} = X_{w,j} \leq W_j, \forall j \in H,\ \forall w \in N^j \tag{3.8}$$

$$W_j \leq \sum_{w \in N^j} X_{w,j}, \forall j \in H \tag{3.9}$$

In order to investigate the effect of including network energy consumption on energy efficient scheduling in data centers, we also formulate and solve another optimization problem Minimization of the Server Energy Consumption (MSEC). MSEC problem is extensively being studied in the literature as a solution for energy efficient data centers. The objective of MSEC is to minimize the energy consumed by servers. The only difference of MTEC and MSEC is the objective function. Both MTEC and MSEC implements dynamic energy management and sleep modes for both servers and network elements. Besides the difference of objective function, MSEC also adopts all of the constraints of MTEC presented in (1-8).

Objective function of MSEC: Minimize the total server energy consumption.

$$\min \sum_{j \in H} W_j P_{server}(j) \tag{3.10}$$

3.2.1. Complexity of the MTEC Problem

The MTEC problem presented in this chapter is a special case of Multi-dimensional Bin packing problem [77]. This problem is also known as Vector Bin Packing problem which is known to be NP-Hard for $d \geq 2$ [77], where d is the number of resource dimensions. Vector Bin Packing problem can be easily reduced to MTEC problem.

Theorem 3.1. *Vector Bin Packing Problem* $\preceq_{APX}$ *MTEC.*

Proof. We can show that Vector Bin Packing problem is a special case of MTEC as follows: Let $\mu_i = 0,\ \forall i$. In other words, we disregard the network elements and route selection. Then, the system is only composed of the servers and tasks waiting for assignment. Let all servers have unit capacity in each dimension, $c_j^l = 1,\ l = 1, \ldots, d,\ \forall j \in H$ where $d = 2$. Then total energy consumption is directly proportional to number of *on* servers. The objective is to assign tasks into minimum number of servers. This special case corresponds to Vector Bin Packing problem where tasks are items and servers correspond to bins. □

Corollary 3.2. *It is proven in [77] that the Vector Bin Packing Problem is NP-Hard. Therefore, due to Theorem 3.1, MTEC problem is also NP-Hard.*

Corollary 3.3. *In [78], it is proven that, there is no asymptotic approximation scheme for vector bin packing problem even for* $d \geq 2$ *unless* $P = NP$*. Hence, due to Theorem 3.1, MTEC problem is also APX-Hard for* $d \geq 2$ *unless* $P = NP$*.*

In [79], it is proved that, there is no asymptotic approximation scheme for vector bin packing problem (and so it is an NP-hard problem) even for $d = 2$. It means that there is no polynomial-time constant-factor approximation algorithms for the MTEC problem, unless $P = NP$. In that regard, MSEC is also APX-Hard. These results motivate the design of efficient heuristics for MTEC problem which is conveyed in Section 3.3.

3.3. Proposed Algorithms

As we have seen in the previous section, MTEC problem is difficult to solve exactly and is even difficult to approximate with a guaranteed factor in general. In that respect, we propose two heuristics namely Best Energy Path Scheduling (BEPS), and Best Resource Fit Scheduling (BFRS). We also introduce Round Robin Scheduling (RRS) as the baseline case in order to evaluate the proposed heuristics.

3.3.1. Best Energy Path Scheduling (BEPS)

Algorithm 3.3 presents a simple which is yet efficient heuristic. At the beginning of each scheduling period, the algorithm starts with checking the finished tasks. According to finished tasks, CPU and memory capacities of servers and remaining link capacities are updated. After capacity update, we check the state of the system elements (servers and switches) and change their state to *sleep* if no task is being served by that server or switch. We adopt immediate sleep policy, which puts elements into sleep mode as soon as they become idle. In the next step, we sort tasks in the task queue according to selected dimension of the task requirement vector (CPU, memory, network). The last step is to find the best path $\mathbf{p}$ for every task in the task queue, from root r to sink s using energy calculation. The corresponding task is assigned to the $\mathbf{p}$, the residual capacities are updated along $\mathbf{p}$. The states of the system elements along $\mathbf{p}$ are checked and switched to *on* state if needed.

In particular, the path cost calculation is based on summing up activation and operations costs on the path. If the selected item along the path is on sleep mode, the cost of the path significantly increases due to high wake up costs. This algorithm runs in polynomial time, i.e, linear in tasks scheduled and number of paths.

3.3.2. Best Resource Fit Scheduling (BRFS)

Algorithm 3.4 presents heuristic based on fitting values of requirements and resources. Different from BEPS, this heuristic determines the best fit server q for each

```
for each time slot do
    check finished tasks and update residual capacities
    update state of the graph elements due to finished tasks
    add newly arrived tasks to the task queue
    sort tasks
    for each task in the task queue do
        p ← the best energy path which satisfies the requirements of the task
        assign task and update residual capacities of server and links along path p
        check and update states of the graph elements along path p
    end for
end for
```

Figure 3.3. Best Energy Path Scheduling (BEPS).

task before finding the best energy path **p**, giving preference to best fit server. We use two best fit measures in our analyses:

(i) Dot-Product: $\sum_i \alpha_l^i c_i^l$. Dot-product maximizes the dot product between the vector of remaining capacities, and the requirements vector of the task

(ii) Norm-Based: $\sum_i (\alpha_l^i - c_i^l)^2$. Norm-based focuses on difference between the resource vectors and the residual capacity under norm 2 [80].

Similar to BEPS, BRFS has polynomial time complexity which is linear in number of tasks scheduled and number of paths.

3.3.3. Round Robin Scheduling (RRS)

The last heuristic we implement is Round Robin Scheduling which is implemented as a benchmark heuristic. RRS is used to show the effect of using smart heuristics. The only difference from BRFS is that this heuristic selects servers in a round robin fashion rather than using a best fit measure. RRS also adopts sleep schedules for both

```
for each time slot do
  check finished tasks and update residual capacities
  update state of the graph elements due to finished tasks
  add newly arrived tasks to the task queue
  for each task in the task queue do
    for each server do
      if server capacity satisfies task requirements then
        calculate fitting value b
      end if
    end for
    q ← server with minimum b
    p ← the best energy path to server q,
    assign task and update residual capacities of servers and links along path p
    check and update states of the graph elements along path p
  end for
end for
```

Figure 3.4. Best Resource Fit Scheduling (BFRS).

servers and network elements.

3.4. Performance Evaluation

In this section, we evaluate the results of our proposed heuristics that we presented in Section 3.3. We evaluate our algorithms by using Google Cluster trace [73]. In this evaluation, we schedule tasks in a time slotted manner. Each slot $t \in 1, 2, ..., T$ where t can be possibly small like a couple of seconds or minutes.

In our system, task requests are composed of computaional requirements and bandwidth requirement. We considered server requirements as CPU and memory. In addition to the server requirements, we consider bandwidth requirement (μ) assigned

to the tasks. However, Google's cluster trace does not include bandwidth requirement of the tasks. So we assume that the bandwidth requirement of $task_i$ is proportional to its memory requirement, $\mu_i = \alpha_i^m \; Gbps$.

In this study, we work on two different DCs explained in Table 3.2. We evaluate the optimal solutions with small DC. In this analyses, the slot length is 1 minute and the evaluation interval is 2 hours. For optimal solutions of MTEC (Opt_{mtec}) and MSEC (Opt_{msec}) problem, we use CPLEX optimization tool. CPLEX can provide efficient solutions to ILP problems. However it takes too much time for CPLEX to converge to optimality for large networks. Hence in the first part of analysis, we evaluate optimal results with small DC.

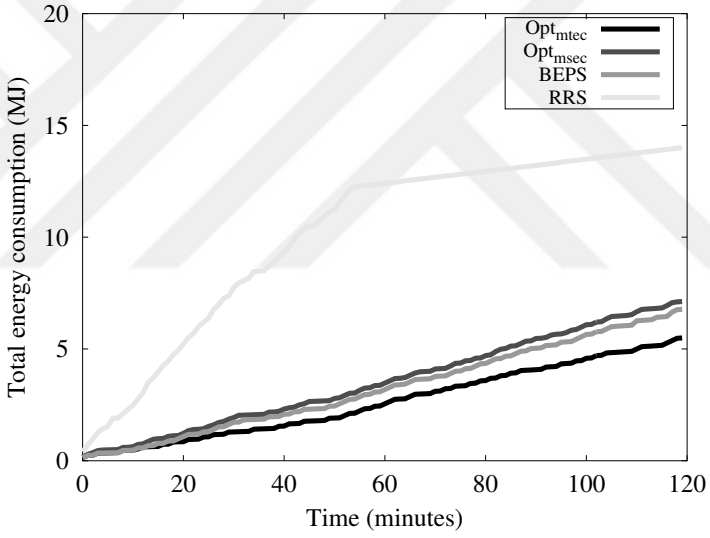
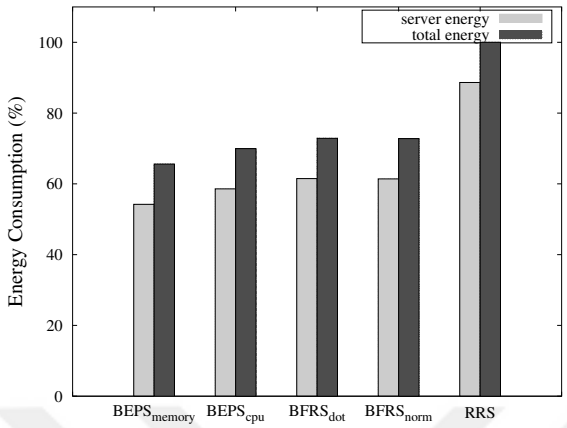


Figure 3.5. Energy consumption of BEPS vs. optimal and RRS.

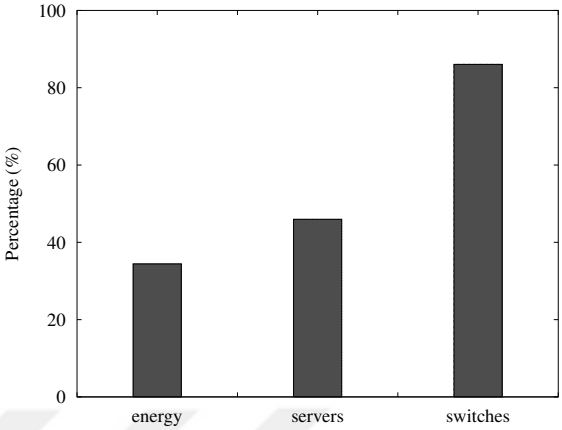
In Figure 3.5, we compare BEPS with optimal solutions, Opt_{mtec}, Opt_{msec} and RRS. Note that RRS implements scheduling of tasks to servers in round robin fashion like most of the current scheduling systems. Hence, RRS constitutes the base case and upper bound where Opt_{mtec} is the lower bound in our analysis. From these analysis, we observe that BEPS performs very close to Opt_{mtec} where RRS consumes significantly high energy by disregarding task colocation.

We extend our analysis in order to evaluate the scalability of our proposed solutions, we use large DC for longer duration. The large DC is composed of 1250 servers

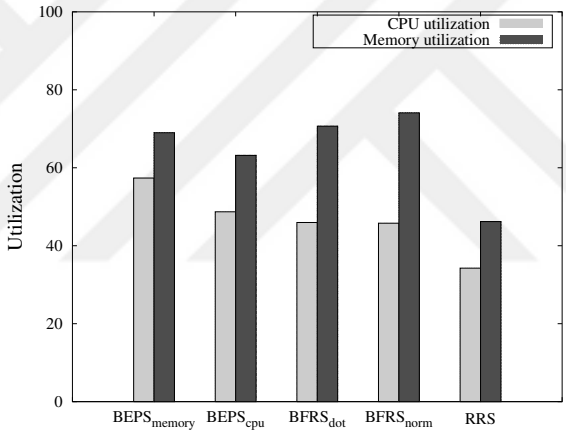
as shown in Table 3.1. Moreover, we adopt shorter slot length as 1 second for fine granularity analysis and the evaluation interval is extended to 24 hours.



(a) Energy consumption.



(b) Energy savings (%) with BEPS.



(c) Resource utilizations with BFRS and BEPS.

Figure 3.6. Total energy consumption, network energy consumption and resource utilizations with BFRS and BEPS and savings with BEPS executing on a heterogeneous cluster of 1250 nodes.

In Figure 3.6(a), we present the energy consumption of variations of BEPS and BFRS. While BEPS$_{cpu}$ and BEPS$_{memory}$ represents the results of BEPS where tasks are sorted according to their CPU and memory respectively, BFRS$_{norm}$ and BFRS$_{dot}$ represents the variations of BFRS with best fit measure dot-product or norm-based as explained in Section 3.3.2. We use RRS as our base case and normalized the other results according to the energy consumption of RRS. The proposed heuristics provide substantial energy savings and perform very close to each other. However, BEPS's

with memory feature gives best results. Moreover, BEPS time complexity is less than BFRS. Specifically, BEPS consumes less than 70% of RRS energy consumption.

In Figure 3.6(b), we show potential savings with dynamic power management by using BEPS compared to *always on* approach. This figure justifies the significant energy savings obtained with BEPS. We can obtain nearly 40% energy savings. In addition, we can put nearly half of the servers and two thirds of the network elements into sleep mode to save energy. Lastly, Figure 3.6(c) shows the memory and CPU usage of *on* servers. High CPU and memory utilization is an indicator of better consolidation of tasks. Similar to the previous figure, in previous figure BEPS_{memory} gives best performance.

3.5. Chapter Summary

In this chapter, we investigate and analyze the impact of a holistic model for the task scheduling with multidimensional resource requirements in heterogeneous data centers. Our proposed model integrates the resource capacity and cost of network segment in addition to the conventional analysis of servers. It is clear that finding the optimal solution for the proposed MTEC formulation is not computationally feasible for large data centers. Hence, we focus on developing efficient heuristics and evaluate our solutions with real data traces. According to our results, nearly half of the servers and network elements can be put into sleep mode in order to save energy. With the proposed heuristics, we achieve better resource utilization as well as energy savings. The limitation of this study is that the scheduler needs exact task requirements for CPU and memory.

4. SCHEDULING FOR WORKLOADS WITH COMMUNICATION NEEDS

Scheduling is a central operation to achieve performance objectives while distributing diversified workloads across heterogeneous resources in an resource efficient manner. MapReduce has emerged as one of the key workloads in today's data centers which alternates between computation and communication intensive phases with bursty workloads. The challenge to make execution of these type of workloads resource efficient, lies in controlling server and network resources simultaneously. The related work offers various good solutions for homogenous systems with the central theme of packing tasks into as small number of servers as possible and thus overlooking the possibility to sleep servers and network components. This study considers a very bursty MapReduce workload with distinct CPU, memory and network requirements executed on heterogeneous data centers, where servers have various CPU/memory capacities and execute request in a process-sharing manner. To reduce energy consumption while maintaining a low task response time, we propose an online energy minimization path algorithm, termed GEMS, to schedule MapReduce tasks, in cooperation with sleeping policies on servers as well as the switches. Using Google MapReduce traces, our simulation experiments show that our proposed solution improves task response times by 35% on heterogeneous data centers and meanwhile gains a significant energy saving of 35% compared to policies which are network agnostic or adopt no sleeping schedule. Overall, we achieve greener and faster MapReduce with (surprisingly) only a slightly higher number of servers, by considering energy consumption rather than conventional approach of considering power values only.

4.1. Motivation

One of the increasingly important class of datacenter workloads is MapReduce application featuring on a powerful and efficient computation of complex queries that involve a huge amount of data. The MapReduce system handles large computations by

splitting them into small tasks and parallelizing their executions across large scale clusters. MapReduce applications alternate between map and reduce phases [68,81], where tasks are processed on distributed computing units and periodically communicate results to the master tasks in an all-to-one fashion. The response time of tasks depends not only on the processing capacities of data centers, but also the network communication that is shown particularly influential for the tail response times [4]. These bursty workload characteristics complicate the challenges of optimizing the performance in a sustainable fashion.

Executing mixed workloads indeed requires a large amount of computation, e.g., CPU and memory, and communication resources, e.g., links and switches. Modern servers and network switches are designed with multiple operating modes, i.e., *on*, *idle* and *sleep* [82], each of which consumes different amount of power and causes varying latencies for task execution. Strategies for achieving green datacenter tend to optimize either energy consumption of servers by leveraging power saving options for individual hardware or intelligent workload execution for the entire cluster. Energy efficiency of data centers is addressed by studies looking from server and network point of view. For the server part, dynamic right-sizing has been extensively studied using analytical approaches [8, 83, 84]. Dynamic right-sizing studies optimize the number of *on* servers at a time according to incoming tasks. These studies focus on instantaneous state of the system, and optimize power consumption by finding the minimum number of servers to satisfy the load. From the network energy point of view, ElasticTree [85] finds the minimum number of active network elements while satisfying QoS requirements. Furthermore, there are some recent studies on virtual machine (VM) placement while considering cost of communicating tasks. A few recent studies [86–88] controlling both server and network try to execute workloads by employing as few number of servers as possible while overlooking the energy saving from *sleep* modes and the degradation of response times due to tight packing of tasks.

Overall, it is hard to control multiple number of resources as well as their operation/power modes for bursty workloads without compromising the response time of tasks. Our study differs from this literature by providing a holistic energy and response

time consideration on a heterogeneous server environment. Our aim is to design an online green scheduler which minimizes the energy consumption while also taking response time into account. Moreover, we consider time aspect in our analysis since our proposed scheduler minimizes energy rather than power consumption at an instantaneous time moment. Furthermore, we introduce a realistic data center environment with heterogeneous servers and switches in our trace driven analysis.

To achieve sustainable and green MapReduce execution in heterogeneous data centers, we propose to control MapReduce task assignments in combination with the sleeping policy of servers and switches. We leverage the idea of finding the minimal energy path on a datacenter topology, where servers and switches are nodes and links are edges with different capacities and power consumptions depending on operation modes. To such an end, we develop an online GrEen MapReduce Scheduler (GEMS), which assigns incoming tasks to paths that can accommodate their CPU, memory, and external/internal network requirements with minimum energy consumption. GEMS activates sleep mode for servers and switches when possible. In addition to realistic power consumption values on a wide range of heterogeneous systems, we employ a task response time model, which includes the delay of switching operation/power modes and the task execution time based on the processor-sharing principle. Essentially, the task response time depends not only on the task requirements but also on the number of colocated tasks.

In particular, GEMS consists of two phases: the first phase finds the minimum energy path that satisfies CPU, memory, and the external traffic requirement of the task and the second phase reserves the bandwidth for paths according to internal all-to-one communication between tasks. Using trace driven simulation, we evaluate GEMS on Google MapReduce trace, one is 4 hour long and the other one is one day long. Our results show that GEMS not only provides significant energy savings but also leads to a noteworthy decrease in response time of tasks, compared to policies that do not put servers and switches into *sleep* mode, and that aim to only optimize server energy consumption. In a nut shell, GEMS achieves sustainable MapReduce execution by allowing a loose packing of tasks onto a slightly bigger cluster and harvesting the

response time reduction due to extra capacity and energy saving by the sleeping mode.

Our contributions can be summarized as follows:

- we employ realistic power consumption values for heterogeneous servers,
- our response time model considers the impact of executing multiple tasks on the same hardware platform, using the scheduling discipline of processing sharing,
- the proposed GEMS controls the number of *on* servers and switches as well as their operation mode,
- our evaluation on traces obtained from production systems show promising gains on energy as well as response time of tasks,
- we achieve energy savings and lower response time by considering energy consumption rather than instantaneous power values only.

The outline of this chapter is as follows. In Section 4.2, we explain the details of Google MapReduce workloads and the heterogeneous data centers. GEMS is described in Section 4.3. The evaluation results are summarized in Section 4.4, followed by the concluding remark in Section 4.5.

4.2. System Model

In this section, we elaborate the complexity of MapReduce workloads, heterogeneous data centers, and different operation/power states of servers as well as switches.

4.2.1. The MapReduce Workloads

To illustrate the workload characteristics and resource requirements of MapReduce applications, we use one day long Google cluster trace [73][1] . The detailed analysis of the trace can be found in [89]. We note that this trace is dominated by MapReduce cluster, meanwhile contain some other applications too. Due to lack of information

[1]The Google trace is dominated by the MapReduce applications. However the applications related details, such as application types and the attributes of map and reduce tasks are not provided.

provided in the trace, it is not possible to differentiate the applications nor the details regarding to MapReduce applications, such as the number of map and reduce tasks. Google trace specifies tasks' CPU, memory requirements, and the corresponding execution time. All task requirements and machine resource capacities are normalized with a range between $[0, 1]$ according to the machine with maximum capacity. However, the network requirement of tasks is not provided in the trace. In order to obtain the network requirement and the communication patterns of tasks, We use Benson et. al.'s [42] real data center network analysis.

4.2.2. Network Requirement

In our system, communication demand of a task is twofold; communication with the master task, and communication with external sources. As for the communication pattern of internal traffic, we assume the tasks of the same job arrive at the beginning of the same slot and one of them is the master task. All tasks belong to same job needs to communicate with master task until task finishes. Bandwidth reservation for internal traffic is made from server which task is assigned to the server which master task is running. So there is an all to one communication for internal traffic. On the other hand, a task needs bandwidth reservation to communicate to an external source during its execution. Bandwidth reservation for external traffic is made from the core switch to the server which task is assigned.

To approximate the network bandwidth demands of tasks, we leverage the network characterization study in [42], which collected the cumulative distribution of flow sizes on real data centers hosting MapReduce applications, as Google trace data does not contain network related information. We propose to estimate internal and external network bandwidth demands per task as their flow sizes divided by the minimum execution time of a task. Here, we approximate the minimum execution time of a task as the CPU demand of a task divided by the server capacity, which the task is assigned to. We note that exact network bandwidth demands depend on the mapping of task-server, whereas the flow size per task is constant. Essentially, for each task, we first randomly draw two flow sizes from the cumulative distribution function (CDF) provided in [42]

for internal/external communication and then calculate their bandwidth demands that are used to reserve link capacity.

In summary, tasks can be assigned to a server, only if a server has sufficient CPU and memory capacity as well as a routing path with sufficient link capacity.

4.2.3. Data Center Architecture

We consider a typical 3-tier data center architecture where each switch is fully connected with the lower layer nodes by links, as shown in Figure 3.1. The first layer is composed of core switches, denoted as l_1, which can be viewed as the source nodes and where the arriving MapReduce tasks enter the data center. The switch nodes in the next two levels are aggregation and access switches, denoted as l_2 and l_3, respectively. The last node layer is heterogeneous servers, denoted as l_4, which have different CPU, and memory capacities. Node layers are connected through links, which have different bandwidth capacities according to the connected layers.

In particular, the datacenter in our study is equipped with 1250 server nodes, 8 core switches, 16 aggregation switches and 512 access switches. The link capacities among core, aggregation and access switches are 10 G, 1G and 1G respectively. The specific values regarding to topologies are listed in Table 4.1. Following Google trace data in [73], we consider four types of server nodes, whose respective percentage and CPU/memory capacities are listed in Table 3.1.

Table 4.1. Topology: number of switches and their power consumption.

		Power [watts]		
Node types	Quantity	P_{sleep}	P_{idle}	P_{on}
Core (l_1)	8	5	16.6	25
Aggregation (l_2)	16	10.2	34	51
Access (l_3)	512	15	50	75

4.2.4. Energy Consumption

Here, we assume all servers and switches are equipped with the technology supporting three operation modes, i.e., *on*, *idle*, and *sleep*. The differences among those states are the power consumption and transition overhead between states. Switching between *idle* and *on* states is automatically triggered by task arrivals and departures and incurs no delay. In contrast to *idle* and *on*, *sleep* mode has much lower consumption but requires (de)activation command. Moreover, a non-negligible overhead in terms of energy and delay can occur when waking up *sleep* servers/switches. Here, we assume extra energy consumption from *sleep* to *on* is computed as the the product of peak power consumption and times duration in such a transition. Moreover, we assume no extra delay and energy consumption are incurred for transition from *on* to *sleep*.

4.2.4.1. Server Energy Consumption. We model the power consumption of an *on* server with a fixed term, i.e., power consumption at the *idle* state, and load-dependent term that increases proportionally with the memory and CPU usage [10] as the following equation:

$$P_{on} = P_{idle} + P_{cpu}U_{cpu} + P_{mem}U_{mem} \tag{4.1}$$

P_{cpu} and P_{mem} are the power values when CPU and memory are fully utilized. U_{cpu} and U_{mem} denote the utilization of CPU and memory, respectively. U_{cpu} and U_{mem} values are between 0 and 1.

The peak power consumption of an *on* server equals to:

$$P_{peak} = P_{idle} + P_{cpu} + P_{mem} \tag{4.2}$$

The energy consumption of server essentially needs to integrate the power consumption over time. Note that particularly, we assume a linear relationship between the CPU/memory capacity and P_{cpu}/P_{mem}, across servers.

4.2.4.2. Network Energy Consumption. Power consumption of a switch directly correlates with the number of ports and line cards [90, 91]. Particularly, each type of switch considered here has a fixed number of ports and line cards. As such, we include power consumption of links to the power consumption of the switches to which they are attached. Taking reference values of the power consumption of switches from [76], we consider power values of core, aggregation, access switches as listed in Table 4.1. The overall energy consumption of data center network sums over all the switches.

4.2.5. Response Time Model

In addition to the detailed model on energy consumption, we also model the task response time defined as time between task arrival and departure. Response time of a task, $T_{response}$ is composed of two terms: T_{wait} waiting time of task before processing starts, and T_x execution time of the task.

T_{wait} depends on scheduling algorithms and wake up delay of servers. In this study, a task is scheduled immediately after its arrival and servers are woken up if needed. We assume wake-up time of servers and switches are 30 sec and 0 sec, respectively. In a special case where tasks are not allowed to queued at servers that do not have enough capacities upon task arrivals, T_{wait} is limited to the 30 seconds. In this study, we only consider algorithms under this special case.

The second contributor of $T_{response}$ is the execution time of the task, T_x. To accommodate the impact of executing multiple tasks on a server at the same time, we propose to scale the execution time of tasks specified in the trace, by the ratio between tasks' CPU requirements and available CPU capacity of server. In particular, we assume the processor sharing principle is used by servers, i.e., co-executed tasks equally share the available capacity.

We illustrate our model by two simple examples. When a task with a CPU requirement of 0.5 is assigned to a server with CPU capacity of 1, its execution time is therefore 1/0.5=2 times faster than the value specified in the trace. When such a

task is assigned to CPU of 1 together with another task with CPU requirement of 0.5, it gets only half of the capacity, i.e., 1/2, and therefore its execution time remains the same as trace. The remaining task execution time is updated at the beginning of each slot according to the excess CPU capacity of the assigned server. As a task can only be assigned to a server with enough capacity to satisfy the requirement, task execution time in our system is bounded by the original execution time specified in the Google trace.

4.3. Green MapReduce Scheduler

In this chapter we propose a novel online algorithm GEMS which assigns MapReduce tasks in a heterogeneous data center with an aim to minimize energy consumption as well as response time. To such an end, GEMS finds the least energy consumption routing path whose nodes satisfy task requirements of CPU, memory and links fulfills external and internal communication. GEMS also controls the operation modes of servers and switches, i.e., enter/wake-up *sleep* modes.

GEMS works in a slotted manner and consists of two phases. In the first phase, GEMS determines task's routing path, γ' and assigns a server to tasks, accommodating CPU and memory requirements and providing link bandwidth for external communication to the outside world. In the second phase, GEMS finds task's routing path, ψ', which reserves bandwidth for internal communication with master tasks, given the task server assignment determined in the first phase. Moreover, GEMS finds path right upon task arrivals and only assign tasks to servers whose capacities are sufficient at the time slot of arrivals. The outline of GEMS is described in Algorithm 4.1.

4.3.1. GEMS Phase I

At the beginning of each slot GEMS starts with checking finished tasks. Once the tasks are finished, the reserved CPU, memory, and link bandwidth are immediately freed. When a server or a switch does not serve any task, it is immediately put into *sleep* mode.

Phase I:

for each time slot t **do**

check finished tasks and update residual capacities

put idle servers and switches to sleep

add newly arrived tasks to the task queue

sort tasks in the ascending order according to memory

for each task in the job queue **do**

compute minimum energy path γ' according to Equation 4.4

assign task and update residual capacities of server and links

wake up servers and switches along γ'

end for

Phase II:

for each task **do**

find minimum energy cost path ψ' to the server which master task is running

establish path and update residual capacities of links

wake up switches along ψ'

end for

end for

Figure 4.1. GEMS (GrEen MapReduce Scheduler).

The next step is to find the minimum energy path γ' for all tasks arriving at the beginning of the slot. The tasks are sorted in an ascending order according to their memory requirements. There are two reasons behind: memory is the typical bottleneck in MapReduce applications and this is indeed observed in our trace. The rationale of sorting tasks in an ascending order is to best take advantage of residual capacities of *on* servers and switches. Sorting tasks in a descending order can easily result in situations where remaining capacities of *on* servers are not sufficient to fit big tasks and a higher number of *on* servers are thus required.

This minimum energy path starts from a dummy root node (all core switches are

connected to root) and ends in a server. Note that, we do not pre-determine the end point, i.e., task-server assignment, before routing is found. Essentially GEMS selects server and path simultaneously while searching for the path. GEMS goes through all paths γ in the system which satisfy the resource requirements of the task and calculates their energy cost and selects the path γ', which has the minimum energy cost. The energy cost calculation for routing path γ, $E(\gamma)$, depends on the states of switches and server nodes along the path, i.e., their operation modes and number of task already in execution.

For each node $i \in \gamma$, its energy cost is composed of two terms, transition energy cost and energy cost of keeping the node running. The transition energy cost is essentially the product of transition power, i.e., peak power (P_{peak}), and the delay before the execution starts, T_{wu}. Note that T_{wu} occurs only when servers wake up from *sleep* mode, and T_{wu} equals to zero changing from *idle* to *on* mode. And, the second term is the energy cost of keeping a node up during the execution of a task. It is calculated as the product of idle power consumption P_{idle} and T_x. We do not include the energy cost of CPU and memory usage (for servers) of a task, as the energy cost of CPU and memory usage of a task is the same on any server, due to the linear scaling across servers' CPU/memory capacity and their power values. T_x is the estimated execution time of the task on a specific server. T_x depends on the assigned CPU capacity, hence number of tasks running on that server. As the energy cost of node i is shared across co-executed tasks, denoted as N_i, we propose to estimate the energy cost of executing a task on node i as:

$$\frac{P_{peak}T_{wu}}{N+1} + \frac{P_{idle}T_x}{N+1}. \tag{4.3}$$

Therefore, one can estimate energy cost of path γ as:

$$E(\gamma) = \sum_{i \in \gamma} \frac{P^i_{peak}T^i_{wu} + P_{idle}T_x}{N_i + 1} \tag{4.4}$$

Essentially, the minimum energy path not only satisfies tasks' demands of CPU, mem-

ory and external communication but also incurs the least amount of energy consumption. When servers and switches on path γ' are in the *sleep* mode, GEMS needs to deactivate them and cause an extra delay in the wake-up processes. After finding the minimum energy path for all tasks, GEMS moves to phase II.

4.3.2. GEMS Phase II

In phase II, GEMS aims to find the minimal energy path, ψ' that sustains internal communication with the master task. In contrast to phase I, the minimum energy path is established between two pre-specified servers. The energy cost calculation for ψ is very similar to γ, except energy cost of server part is not considered. When a task is finished, it is removed from the corresponding server and the bandwidth reserved on communication paths γ' and ψ' is released. The complexity of this algorithm depends on the complexity of the energy path calculation. The algorithm operates in $O(PN_w)$ hence it is in the order of tasks waiting to be scheduled (N_w) and number of paths in the architecture (P).

4.4. Performance Evaluation

To evaluate the effectiveness of GEMS on a heterogenous datacenter, we use a large scale MapReduce trace by Google [73] and build a trace driven simulator. In particular, we evaluate a datacenter of 1250 server nodes, whose capacities and power consumption are listed in Table 3.1. This datacenter is connected by a 3-tier fat tree like network, whose topology and power consumption are detailed in Table 4.1. We present the results of 4 and 24 hour long traces.

The aim of our evaluation is two-fold. First, we seek to show the advantages of activating *sleep* mode of servers and switches for bursty MapReduce workloads. Second, we emphasize the the energy savings and response time reduction that are resulted from considering server and network consumption simultaneously. The metrics of interests are, the average task waiting time (T_{wait}), average task response time ($T_{response}$), system utilization, and, most importantly, energy consumption of entire

Table 4.2. Performance evaluation results of 4 hours.

	4 HOURS		
metrics	GEMS	GEMS(noNetwork)	GEMS(noSleep)
E_{total} [kWh]	806	968	1258
E_{server} [kWh]	764	773	1125
$E_{network}$ [kWh]	42	195	133
T_{wait} [sec]	18	24	0
$T_{response}$ [sec]	128	145	198
average U_{mem} [%]	56	64	63
average U_{cpu} [%]	68	75	78
average *on* servers	340 (max 1248)	341 (max 1224)	316 (max 850)
average *on* switches	12 (max 13)	533 (max 536)	15 (max 17)

system (E_{total}), server part (E_{server}) and network part ($E_{network}$), measured in kWh. The summary of evaluation results is presented in Table 4.2, Table 4.3 and discussions are detailed in the following subsections.

4.4.1. Comparing with noSleep Policy

To evaluate the pros and cons of using *sleep* mode in GEMS, we implement a "noSleep" version for GEMS algorithm, i.e., GEMS(noSleep). The difference between GEMS and GEMS(noSleep) is, when a switch or server is not serving any task then it enters immediately into *idle* mode rather than *sleep* mode. As a result, GEMS(noSleep) incurs no waiting time T_{wait} or energy cost for transition from *idle* to *on* mode. Hence, the computation of minimum energy path, $E(\gamma)$ and $E(\psi)$, are different in both cases. Nevertheless, *sleep* mode has much lower power consumption than *idle*, for all server and switch types considered in Table 3.1 and Table 4.1.

As shown results in Table 4.2, GEMS indeed results into a much lower energy consumption, in terms of total , server and network, compared to GEMS(noSleep). And,

Table 4.3. Performance evaluation results of 24 hours.

	24 HOURS		
metrics	GEMS	GEMS(noNetwork)	GEMS(noSleep)
E_{total} [kWh]	4278	5107	6428
E_{server} [kWh]	4068	4125	5763
$E_{network}$ [kWh]	210	982	665
T_{wait} [sec]	18	23	0
$T_{response}$ [sec]	138	162	223
average U_{mem} [%]	60	64	66
average U_{cpu} [%]	71	77	82
average *on* servers	409 (max 1248)	409 (max 1224)	376 (max 943)
average *on* switches	11 (max 13)	534 (max 536)	15 (max 17)

GEMS incurs a non-negligible delay, $T_{wait} = 18$ sec, due to waking up *sleep* servers. Surprisingly, GEMS is able to achieve lower task response times than GEMS(noSleep). To further contrast GEMS and GEMS(noSleep), we also summarize the percentages of energy savings and of reduction in task response times in Figure 4.2. Clearly, GEMS has nearly 30% server and 65% network energy savings via adopting *sleep* mode for servers and switches. We reason the prominent energy saving from sleeping switches is due to the bursty workload. As for average task response time, GEMS achieves 128 second per task, accounting for 35% reduction, even though there is an extra delay due to waking up *sleep* servers.

4.4.2. Comparing with noNetwork Policy

To address the importance of optimizing network as well server energy for MapReduce workloads, we compare GEMS with a green scheduler that only tries to optimize the server energy, called GEMS(noNetwork). For a fair comparison, GEMS(noNetwork) also uses same sleeping policy as GEMS, i.e., servers and switches enter *sleep* mode right after completing tasks. GEMS(noNetwork) assigns tasks to servers, by only con-

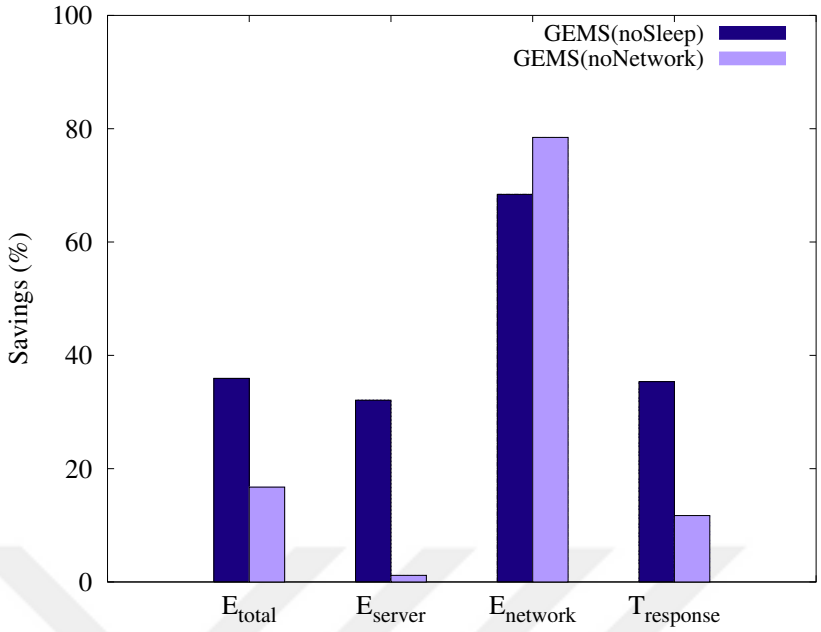


Figure 4.2. GEMS savings compared against GEMS (noSleep) and GEMS (noNetwork): executing MapReduce on a heterogeneous cluster of 1250 nodes.

sidering their CPU/memory capacities and the energy cost of servers, i.e., only the destination node of minimum energy path, γ. As for the task external and internal communication, GEMS(noNetwork) randomly finds internal and external communication paths.

From Table 4.2 and Table 4.3 , GEMS(noNetwork) indeed consumes slightly more server energy than GEMS, however consumes much higher network energy. As a net result, GEMS still consumes less total energy than GEMS(noNetwork), i.e., roughly 15%, shown in Figure 4.2. However, the average task response time of GEMS(noNetwork) for 4 hour trace is 145 second that is roughly 12% higher than the GEMS result, 128 second. Such an observation is due to that GEMS(noNetwork) tries to assigns tasks more tightly into servers. Consequently, the execution times of tasks increase tremendously and only little energy saving from the server part is achieved. On the contrary, GEMS carefully chooses communication path and assign fewer number of tasks into servers. GEMS results into shorter task execution times, which in turn bring advantages in saving energy. We note that such a benefit is often overlooked in related studies that do not model task response times.

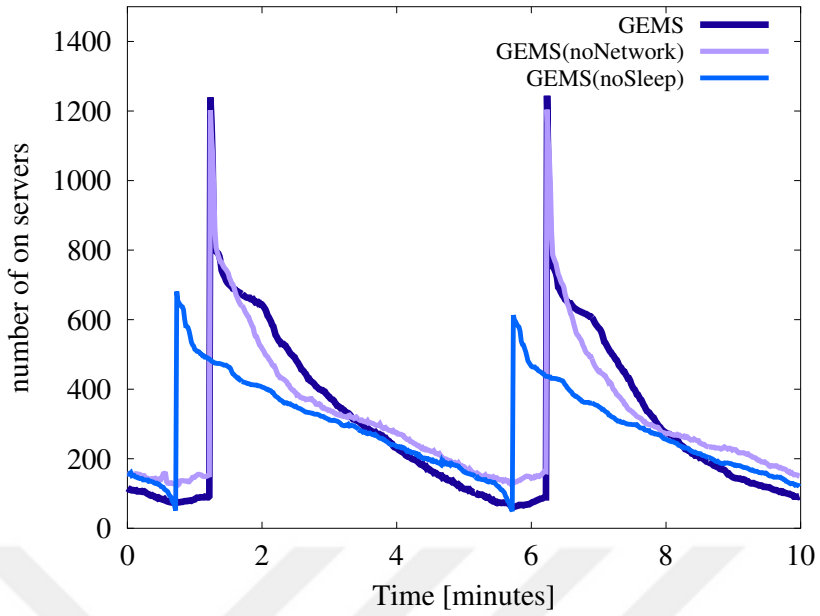


Figure 4.3. Number of *on* servers executing MapReduce on a heterogeneous cluster of 1250 nodes.

4.4.3. Scalability Analysis

To see the scalability of GEMS in a larger scale, we evaluate GEMS against GEMS(noSleep) and GEMS(noNetwork) under a 24 hour trace. Similar to 4 hour results, GEMS achieves significant energy savings as well as response time reduction.

Figure 4.3 shows a 10 minute snapshot of the number of *on* servers changing over time with all three algorithms. One can see that as GEMS(noNetwork) and GEMS apply sleeping, their response to bursty workload is shifted and appears roughly 30 sec after the arrival of the burst, because of the 30 sec wake up time of servers. Moreover, GEMS is more sensitive to bursty workloads and activates many more new servers. Due to the high number of server provisioning during the burst, tasks are completed faster and therefore the number of *on* servers can drop dramatically after the arrival of a burst. Overall, GEMS is able to provide a higher number of servers for the bursty workloads promptly and effectively sleep servers and switches during the low loads, so that both response times and energy consumption are minimized.

4.5. Chapter Summary

Motivated by significant energy draw of MapReduce workloads on today's datacenter, we develop GEMS, a green MapReduce scheduler. GEMS aims to minimize energy consumption of servers as well as network components by finding minimum energy path for tasks according to their CPU, memory and network requirements and adopting the *sleep* mode. Overall, GEMS tries to provide a sufficiently high number of servers for the bursty workloads while effectively running servers and switches in sleep mode during the low system loads. Using on-production Google MapReduce traces, our simulation results on a heterogeneous cluster of 1250 servers show that GEMS is able to achieve significant reductions on average task response times and energy consumption compared to schedulers that overlook sleeping policy or energy spent in networking. In contrast to conventional approaches that focus on instantaneous power consumption, GEMS considers energy consumption, i.e., the product of power consumption and response times. The efficiency of GEMS lies at promptly providing a higher number of servers during the peak loads, thus leading to a significant reduction on the response time of tasks as well as the cluster energy consumption.

5. RESOURCE INEFFICIENCIES OF LARGE CLUSTER SCHEDULERS

Scheduling is a central operation to achieve "green" data centers, i.e., distributing diversified workloads across heterogeneous resources in an energy efficient manner. Taking an opposite perspective from most of the related work, this study reveals the "brown" side of scheduling, i.e., wasted executions (so called brown resources). Based on real trace analysis, we pinpoint the dependency between priority scheduling and task eviction that causes wastes resources and present a brief characterization study focusing on workload priorities.

In this study, our objective is to better understand the brown side of data center scheduling and further mitigate the resource inefficiency of the scheduling, using a field trace provided by Google. To such an end, we focus on a particular scheduling event in Google trace – task eviction – which is triggered by the scheduler due to task congestion, reservation excess or hardware failure. The first step of our study is to qualitatively identify its dominant root cause by analyzing the sequence of tasks co-executed on the same server. Our finding that priority scheduling is the main cause of eviction leads us to conduct a workload characterization analysis of different priorities, including interarrival times, CPU and memory demands of tasks. To quantify the degree of brown resources due to the priority scheduling, we further develop a simulation-based analysis that models the cluster as parallel and heterogeneous servers configured with a fixed number of slots and cores which is explained in details in Chapter 6 and Chapter 7.

5.1. Google Cluster Trace

The Google cluster trace [33] represents a rich heterogeneous workload mix, particularly MapReduce, on a large heterogeneous cluster for 29 days. The focus of the related trace analysis is on providing an overall view of statistical properties about all scheduling events [3, 31].

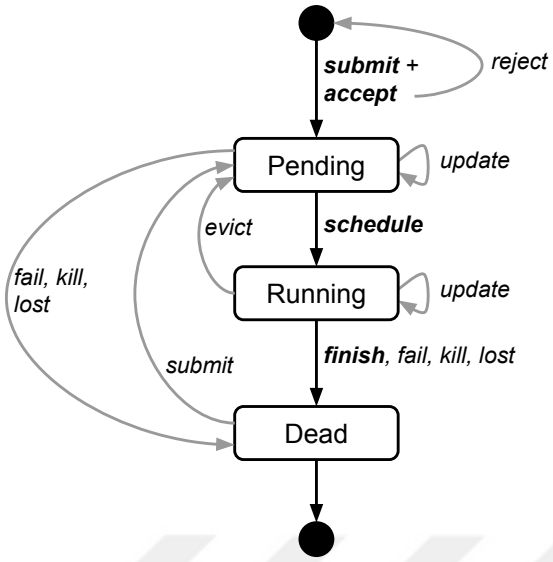


Figure 5.1. The task state transition diagram of Google Scheduler [2].

Figure 5.1 shows the states of a task starting from arrival until its depart from Google's system [2]. After a task is scheduled and started running, it can be evicted and the evicted task is returned to pending state. In particular, our analysis is based on eviction events which are recorded in the Task Events table, the Task Usage table and the Machine Events table of the trace, dated between midnight of 09-05-11 and midnight of 16-05-11. As evicted tasks are automatically resubmitted to the scheduler, tasks can experience multiple evictions before leaving the system with an (un)successful finish. Combining the aforementioned tables, we know the CPU and memory demand of each task, as well as the execution sequence of tasks on the same server. Typically, tasks request a certain amount of resources, i.e,. CPU, RAM and DISK specified in the range of normalized values [0 1], when they are submitted. Moreover, tasks are associated with priorities, ranging between [0, 11], where higher values represent higher priority class tasks.

5.1.1. Workload Heterogeneity

Typically large computing clusters receive several requests with different priorities, performance objectives and resource demands [3, 34]. The Google trace consists of 12 priority classes. The priority classes are grouped as production (9–11), middle (2–8), and gratis (0–1) classes in the trace documentation, where production classes

have the highest priorities.

All task resource demand, usage and machine resource capacities are normalized with a range between $[0, 1]$ according to the machine with maximum capacity. Table 5.2 shows us the normalized CPU demand, memory demand per task. In contrast to usual cases, in Google trace high priority tasks require more resources than low priorities. The variability of resource demand varies significantly, e.g. priority class 11 tasks have an average CPU demand which is 97 times higher than the class 0 while its arrival rate is really low. According to our analysis on Google Cluster trace, we observe that high priority tasks (production classes) arrivals are rare but they occupy more resources and execute longer than low priority classes. These properties make scheduler design more challenging in order to prevent starvation of low priorities while giving preference to high priority classes.

5.2. Analysis of Eviction

Motivated by the fact that the number of evicted tasks is non-negligible and eviction events are highly controlled by the scheduler, we first try to understand the root causes of evictions. Furthermore, we analyze workload characteristics, in terms of arrival patterns and resource demands with respect to the root causes.

5.2.1. Cause of Eviction

Although the trace document provides a textual explanation of why eviction takes place, there is no ready information or explanation for individual eviction. As a result, we categorize every eviction based on certain causes, summarized by the trace document and our evaluation criterion, and identify the most critical one.

According to [92], the following events can cause eviction: (a) machine failure; (b) arrival of higher priority tasks; (c) disk failure; (d) reservation excess, where the total requested resources of servers are greater than their capacities. As the trace doesn't contain information related to disk failures, we are unable to analyze their

Table 5.1. Causes of eviction and their percentages.

Cause	Tolerance (τ)	Percentage
Machine failure	$\{0, 15, 30, 45\}$ s	0.67%
Memory excess	0	1.74%
Disk excess	0	0.0002%
Higher priority task	1s	93.69%
Higher priority task	5s	93.76%
Higher priority task	10s	93.94%
Higher priority task	20s	94.59%
Higher priority task	30s	95.36%

impact on eviction. Analysis of the reservation excess requires complete information about the requested resources of all the co-executed tasks at eviction time, which we are unable to find for all the evictions due to the particular trace format and the truncated observation window. Consequently, we inspect a fifth event, task resource overcommitment, for the situation where the actual used resources per task are greater than the requested resources upon submission time.

For each task eviction, we try to inspect if each cause withstands under multiple thresholds. For example, we count how many eviction records are present in the trace within 0, 15, 30 and 45 seconds from a machine failure. Note that a single eviction can be caused by multiple events. In the rest of the section, we analyze in order, the relevance of the following causes: machine failure, task resource overcommitment and preemption due to higher priority tasks. We summarize our results in Table 5.1.

5.2.1.1. Machine Failure. To correlate machine failures with eviction, we compute the downtime interval of each machine, and compare them with each eviction time stamp. We identify their dependency by a time threshold τ, to account for potential profiling delay. As long as the time stamp of an eviction is τ seconds after the start of the machine downtime interval, we consider it as caused by machine failure. We apply

multiple values for τ, i.e., 0, 15, 30, and 45 seconds, and obtain the same results for all of them, i.e., only 0.67% of eviction is caused by machine failure. Therefore, we conclude that machine failure is not a main cause of eviction.

5.2.1.2. Task Resource Overcommitment. Here, we try to understand if tasks are evicted due to consuming more resources than the ones they request at submission time. We note that, according to the trace document [92], tasks with resource overcommitment should be terminated by the scheduler using kill event. Surprisingly, we find that some eviction events can also be related to task resource overcommitment. In particular, we focus on memory and disk and overlook CPU, which is allowed to use spare capacity as clearly stated by the documentation. For each eviction, we determine whether the maximum memory or mean disk usages are greater than the requested ones, by extracting this information from the Events and Usages tables. We use the mean usage of disk because the maximum disk usage is not provided by the trace.

Results summarized in Table 5.1 show that only a very low percentage of eviction is related to memory excess, roughly 1.74%, and a negligible percentage of eviction is related to disk excess.

5.2.1.3. Preemption by Higher Priority. We propose an intuitive approach to see if higher priority tasks cause eviction, by comparing the time stamp of eviction and the scheduling time of higher priority tasks on the same machine within a time threshold of 1, 5, 10, 20 and 30 seconds. In the following, we identify with the term "evictor" the task which is scheduled on the machine, whereas we name "evicted" the descheduled task. We first query the Events Table and extract all the eviction records, then, for each of them, we search for potential evictor tasks considering three criteria, i.e., time stamp, priority and the machine ID in the Events table. In particular, a task is classified as evictor when the following condition holds: $t_{KO} \leq t_{KI} \leq t_{KO} + \tau$, where t_{KO} denotes the time stamp of eviction, t_{KI} is the scheduling time of the evictor task and $\tau \in \{1, 5, 10, 20, 30\}$ seconds.

As shown in Table 5.1, we can identify that roughly 93% of eviction has corresponding evictor tasks with higher priorities, for all the threshold values considered. In contrast to lower percentages of other causes for eviction, scheduling of high priority tasks on the same machine appears to be the most relevant and dominant cause.

5.2.2. Eviction per Priority

Motivated by the significance of priority in the task eviction process, we present the distribution of evictions across different priorities, particularly the relationship between the evictor and evicted task. Table 5.2 summarizes the distribution of tasks and evictions across different priorities. For the latter, we also report the average of the number of evictions experienced by a task in all priorities considered.

In terms of evicted tasks, priority 0 and 1 account for 92.7% and 6.3% of evictions, respectively. The number of evicted tasks decreases exponentially when the priority increases. Moreover, the mean number of evictions experienced by tasks are particularly high for priority 0 tasks. This means that tasks from priority 0 are not only evicted frequently, but also experience repetitive evictions during their execution. As for evictor tasks, preemption is mainly caused by priority 4 tasks. Indeed, when looking at priority pairs of scheduled and descheduled tasks, 70% of them are composed of priority 4 tasks preempting priority 0 ones. Another fact worth noting is that only 38.09% of the evicted tasks across all priorities could successfully complete their execution in our one week trace. Essentially, CPU resources due to the execution of those re-evicted and unsuccessful tasks are wasted, i.e., the brown resources. This low percentage highlights that evictions have a strong negative impact on the task success rate and cause brown resources.

Overall, the distribution of evictions across priorities clearly emphasizes the central role of priority in the eviction process, where low priorities can be descheduled frequently and result in high failure rates and brown resources, whereas high priorities are rarely preempted.

5.2.3. Workload Analysis of Priorities

We present the main characteristics of different priorities, which further drive the brown analysis in the following simulation section. We consider, individually, each priority along two dimensions: (1) arrivals pattern and (2) used resource demands. Particularly, we only consider resource demands of tasks arriving to and departing from the system in our one week observation window. For the arrival patterns, we study the task interarrival time of each priority in the system. To better quantify the impact on physical resources, we use a metric called resource demand, defined as the product of task service demand and the amount of resources used by tasks. We consider two different types of resources, i.e,. number of cores and memory, and use separate units of measurement for each resource demand, i.e., $CPU \cdot sec$ and $RAM \cdot sec$, respectively.

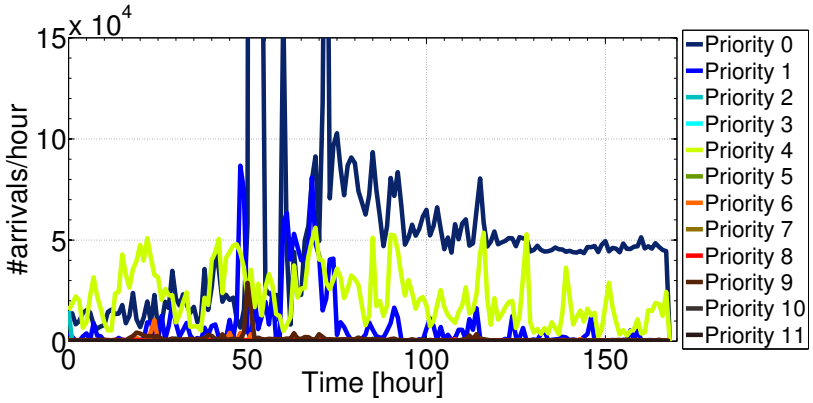


Figure 5.2. Task arrivals of different priority classes from Google Trace.

We summarize the mean values of task interarrival time and resource demand for each priority in Table 5.2. Interarrival times vary significantly across priorities. Arrivals at priorities 0, 1 and 4 are very frequent. Indeed, these priorities cover the 96.01% of the tasks observed in one week, as shown by the second column of the table. On the contrary, priorities 3 and 5 rarely experience new arrivals, resulting in high interarrival times. Higher priorities do not necessarily correspond to higher interarrival times, as one can see from the table. Moreover, we note that the hourly arrivals rate for each priority highly fluctuates in all priorities. In particular, priority 0 experiences the highest peaks of hourly arrival rates, reaching up to 1.1 million arrivals per hour.

In terms of used resource demands, high priority tasks clearly use more physical resources than lower priorities. As one can see from Table 5.2, priority 11 tasks have, on average, a CPU demand which is 50 times higher than the overall mean (reported at the bottom of the table) and their average RAM demand is 73 times higher than the global mean RAM demand. Low priorities, instead, show lower resource demands. Overall, tasks request more CPU demand than RAM, especially at low priorities. In summary, our analysis shows that high priorities have higher impact on physical resources than lower priorities, whereas arrival pattern and interarrival time are not affected by the task priority.

Parallel to the resource demands, high priority class tasks average resource usages are also higher than low priority tasks average resource usages. Overall, tasks require more CPU than memory, especially for low priorities. Moreover, we note that the hourly arrivals for each priority highly fluctuates for all priorities, see Figure 5.2. The arrival rate is clearly highly fluctuating regardless of the priority, as the number or task submissions in consecutive hours can easily vary of 40000 units. In particular, priority 0 experiences strong and swift peaks in the hourly arrival rate, reaching up to 1.1 millions of arrivals per hour.

In summary, our analysis shows a contrast to the assumed usual cases. In the Google trace high priority tasks require more resources than low priority tasks which makes the scheduler design quite challenging. The scheduler must prevent starvation of low priorities tasks while giving priority to the high priority tasks at the same time.

5.3. Chapter Summary

In this chapter, we provide a quantitative analysis of the resource waste due to the scheduling policy, in particular the priority scheduling. Our analysis is based on a large scale cluster trace provided by Google and a trace driven simulation that embeds detailed models of a system scheduler and response times for different priority tasks. Our key findings are that priority scheduling causes a significant number of evictions and that higher priority tasks result in high resource demand. Using a trace driven

simulation, we show that there is a significant amount of brown resources associated with basic priority scheduling policies in the following chapters.

Table 5.2. Percentage of tasks, percentage of evictions, interarrival time, resource demand and resource usage breakdown by priority.

Priority	% of tasks	% of evictions	Interarrival time [s]	CPU demand [cpu·s]	Memory demand [ram·s]	CPU usage	Memory usage
0	23.29	92.73	0.059153	15.408	9.1068	0.0066832	0.0035707
1	12.78	6.26	0.43635	40.758	33.47	0.012176	0.0049097
2	0.39	0.36	14.586	382.12	204.4	0.018668	0.0061447
3	≤ 0.01	0	17876	0.24663	0.025122	0.0034657	0.00036929
4	59.94	0.41	0.16401	49.768	27.721	0.019338	0.0064713
5	≤ 0.01	0	30601	0.15221	0.023401	0.0034661	0.00041077
6	1.28	≤ 0.01	7.5633	12.628	4.6929	0.020213	0.00298
7	≤ 0.01	0	2243.9	n/a	n/a	n/a	n/a
8	1.10	0.04	9.0927	25.975	39.625	0.0037737	0.003922
9	1.19	0.19	2.9024	269.47	230.65	0.0071297	0.0042992
10	0.01	≤ 0.01	86.006	66.186	85.448	0.0092705	0.0042258
11	0.01	0	1144.3	1506.8	1413.6	0.010355	0.014984
All	100	100	0.038519	30.236	19.249	0.010376	0.0044162

6. A FLEXIBLE, HOLISTIC SCHEDULING FRAMEWORK FOR LARGE CLUSTERS

In order to meet the performance objectives of diverse workloads, schedulers rely on evictions even resulting in waste of resources due to lost executions of evicted tasks. It is not straightforward to design priority schedulers which capture key aspects of workloads and systems and also to strike a balance between resource (in)efficiency and application performance tradeoff. To explore large space of designing such schedulers, we propose a trace-driven cluster management framework that models a comprehensive set of system configurations and general priority-based scheduling policies. In particular, we focus on the impact of task evictions on resource inefficiency and task response times of multiple priority classes driven by Google production cluster trace.

6.1. Motivation

A common approach is to execute different applications on separate clusters dimensioned for the peak load in order to meet application specific service level objectives. Consequently, the systems suffer from low resource utilization. To improve the system resource efficiency, hosting multiple types of applications on the same cluster is often sought but meanwhile the system complexity, particularly the complexity of the schedulers greatly increases [6].

Many different schedulers have been proposed in the literature [8, 93, 94] with the main focus of increasing the energy efficiency by consolidating workload and minimizing the number of active servers. Most of them are based on rather homogeneous systems and workloads, i.e., servers with the same capacities, tasks with similar resource demands and priority. However, workloads executed on real systems are highly heterogeneous and have diversified resource usages, service level objectives (SLOs) and class privileges [3, 47], makes existing solutions suboptimal. Moreover, to meet the SLOs under heterogeneous workloads, schedulers rely on task evictions. Evictions are often overlooked by prior work although it is a major source of resource inefficiency.

Task evictions are controlled by the scheduler which occur due to task congestion, reservation excess or hardware failure. The execution progress of task can be suspended or lost at the time of eviction depending on the system. In particular, non-resume systems that do not preserve the task state during eviction suffer from significant wasted resources under bursty arrivals. Overall, it is hard to design scheduling policies that capture the complex system and workload characteristics and optimize the tradeoff between response times and resource (in)efficiency due to evictions.

Motivated by the lack of a framework to analyze the system performance with different scheduling policies under highly heterogeneous workloads and evictions, we propose a new cluster management framework. Our objective is twofold:

(i) to provide a platform to analyze and better understand the system behavior under different workload conditions, system settings, scheduling policies and eviction strategies that make it easier to propose new energy-efficient systems with improved task response times;
(ii) to quantify the impact of evictions on system performance and resource efficiency.

In the subsequent sections, we investigate the main causes of resource "inefficiency" in current systems in order to motivate the design decisions.

6.1.1. Impact of Evictions

We first pinpoint the impact of priority scheduling and task eviction on wasted resources using the Google Cluster trace [33], which contains a rich heterogeneous mix of workloads running on a large heterogeneous cluster for 29 days. For more details about the trace, we direct the interested readers to characterization studies [3, 95] presenting overall task scheduling statistics.

As presented in Chapter 5, priority arrival is the main cause for eviction and the insufficient resource, such as memory, contributes to a small number of eviction events. Our 7-day trace analysis identifies two main causes for eviction. Across all

evictions, around 95% are priority evictions and 2% are memory evictions, the rest of the evictions occur either due to machine failure or disk excess. Evicted task can be rescheduled, however we observe that more than 43% of evicted tasks experience subsequent evictions, i.e., are evicted at least twice. As a result, only 38.09% of the evicted tasks across all priorities could successfully complete their execution. The low percentage of successful executions demonstrates the strong negative impact of evictions on the task success rate which in return leads to wasted resources. Indeed, all resources spent on an unsuccessful task execution are wasted. The central role of prioritization is clearly emphasized by the distribution of evictions across priorities. Low priorities are evicted more frequently which resulting in high failure rates and wasted resources, on the other hand high priorities are rarely preempted.

Motivated by the significant negative impact of evictions in order to better understand and reduce the resource inefficiency of priority scheduling, we propose detailed eviction models presented in Section 6.2.3.3 and Section 6.2.3.4.

6.1.2. Impact of Overbooking

Another significant cause of "inefficiency" is resource overbooking where the scheduler allocates resources according to user set task requirements. However, users usually overestimate the resource usage of tasks. Hence, the cluster looks full even though the actual usage is far below the resource reservations. In this case, the illusion of a full cluster triggers unnecessary evictions which affect the system negatively. The Google Cluster trace analysis shows a very heavily overbooked system. The total resource reservations at almost any time accounts for more than 80% of the cluster memory capacity and more than 100% of the cluster CPU capacity [3]. However, the measured overall usage is much lower: averaging over one-hour time windows, memory usage does not exceed 50% of the cluster capacity and CPU usage does not exceed 60%. Users usually overestimate their resource requirements as shown in Figure 6.1. Also users want to guarantee their successful service execution by over-provisioning. This is because, if a task exceeds its resource requirements, it is automatically evicted by the scheduler. Hence, Google clusters are usually fully booked even though the actual

resource utilization is only half of the resource reservations.

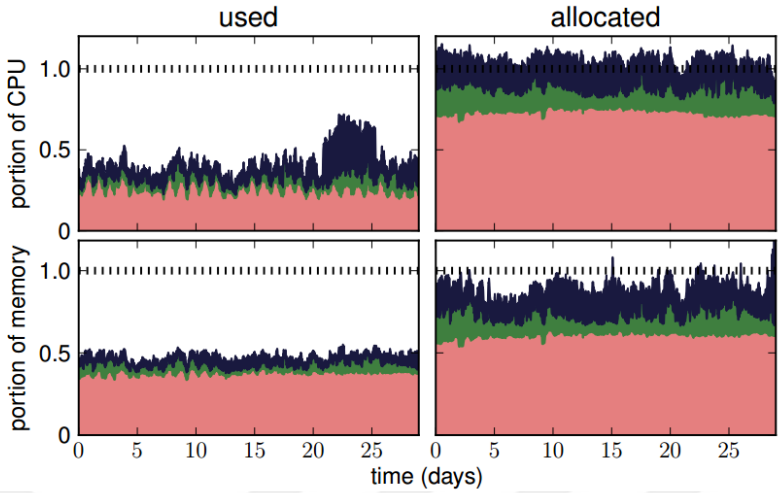


Figure 6.1. Average resource reservations versus actual usage in Google cluster trace (moving hourly) [3]. Production, middle and batch classes from bottom to top.

To overcome this problem we use a dynamic, elastic resource allocation scheme namely slot-based resource assignment where the number of tasks concurrently running on a server is limited by the number of slots. The slot-based system has the advantage that it does not require user defined resource demands. Furthermore, we do not specifically limit the resource usage of individual tasks, hence our approach does not suffer from resource fragmentation and overbooking. The details of the resource allocation schemes are explained in Section 6.2.3.1.

6.1.3. Contributions

We first analyze the Google Cluster trace to better understand the challenges in order to develop an effective cluster scheduler. Due to lack of public information about the insights of Google Cluster scheduler, we turn our attention to trace analysis in order to investigate the inefficiencies and extract some working principles of the Google scheduler. The most remarkable properties we reach from the trace analysis are: significant amount of resource inefficiency and workload heterogeneity, which constitute the main motivations of this study. Due to the complex reciprocal dependency of events and system dynamics, it is nearly infeasible to quantitatively infer the main contributors of evictions without a simulation framework.

According to these findings, we propose a cluster management framework which is designed to quantify and minimize the inefficiencies discovered in the Google Trace. This framework also incorporates many complex design parameters that enables exploring the design space of scheduling policies with a particular focus on the impact of task evictions. By utilizing this framework, we propose and evaluate several scheduling designs and eviction policies and discover the impact of prioritization, evictions and workload characteristics. Hence, we present a comprehensive set of experiments of priority scheduling and evictions on large computing clusters with various tunable parameters and policies.

The contributions of this study can be summarized as follows:

(i) We introduce a new cluster management framework which provides the control of response time per priority class by priority scheduling.
(ii) We demonstrate the importance of eviction policies in order to improve response time and resource efficiency for highly heterogeneous workloads.

In this chapter, we propose a resource and workload-aware cluster management framework, which provides system performance quantification for different system settings and scheduling policies. In Section 6.2, we describe the system properties and eviction policies in details. Then, we characterize the workload and server heterogeneity of the trace in Section 6.3. We investigate the system performance under different design decisions in Section 6.4.

6.2. System Model

At the heart of our framework lies a cluster system simulator, which is capable of capturing the energy consumption and response times of complex workloads under various what-if scenarios. The system consists of two main parts: the scheduler and a set of servers. Incoming tasks are enqueued at the scheduler and dispatched to available servers in a time-slotted fashion. The tasks are processed by a set of servers, where the server environment is consisting of heterogeneous multicore machines. Since evictions

are the main cause of wasted resources, as investigated in Chapter5, we model both priority and memory evictions which represent two main eviction types observed in real systems. Moreover, the scheduler also controls the power state of each server by putting unused servers to sleep in an attempt to save energy [82, 96].

Figure 6.2 presents an overview of the system options for the scheduling policies, server configurations, system configurations and eviction policies supported. In the following subsections, we describe them in detail together with energy consumption and response time models.

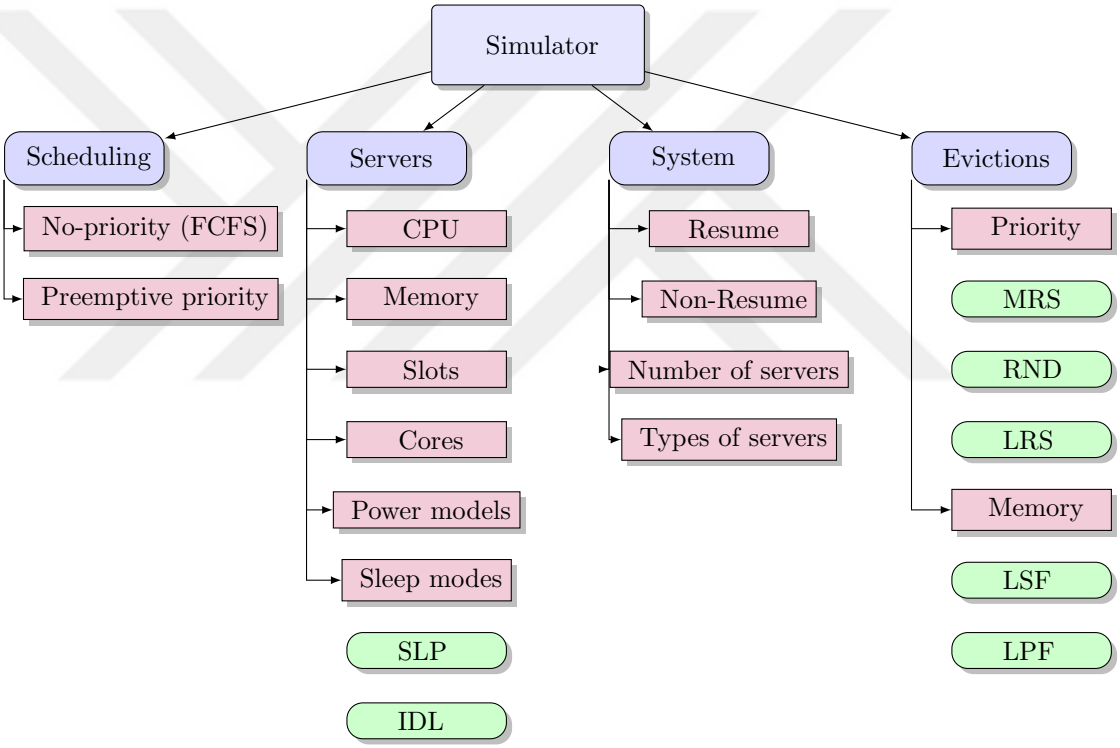


Figure 6.2. Main options of the simulator framework.

6.2.1. Task Model

Each task has a set of attributes. In our system model, we assume each task is characterized by the following attributes: the arrival time, CPU demand, average memory usage, and priority class. These task attributes can be generated synthetically or taken from a trace, either one is supported. We use the arrival time to simulate the task arrival, priority class at the time of scheduling, and resource demand to decide on the execution time (finish) of the task. Also, our system does not require the resource

demand information at the time of scheduling.

6.2.2. Server Model

We provide a heterogeneous multicore server environment, where each server can hold different number of CPU/cores and different amount of memory. Each server holds a predefined number of slots and each slot can be used by only one task. The number of slots for each server can be set differently based on its resource capacities. Task assignment is done based on a slot-based resource assignment scheme to increase resource utilization, where the details are explained in Section 6.2.3.1.

Each task has specific CPU and memory demands. The share of resources per task is computed as follows. On one hand, the server CPU is equally shared across the tasks running on the same server. Since a task occupies exactly one slot, the maximum assigned CPU rate is the core capacity as the upper limit. And the minimum assigned CPU capacity is determined by the CPU capacity divided by the number of slots. On the other hand, the memory is assigned based on the average memory usage of the task. If the total server memory is exceeded, memory evictions occur as described in Section 6.2.3.4.

6.2.3. Scheduler Model

We employ a priority scheduler which is preemptive and resources are allocated according to the slot-based resource assignment model. The tasks are assigned to servers based on server availabilities and capacities, and task priority class without requiring any information about task resource requirement, usage or demand. The design decisions of the scheduler are summarized in Table 6.1 and the details of the working principles of the scheduler is explained in the following sections.

6.2.3.1. Slot-based Resource Assignment. Current approaches often use user-provided resource requirements during task assignment. This approach often suffers from re-

Table 6.1. Design decisions of the scheduler model.

Feature	Design decision	Section
Resource Assignment	slot-based no a priori resource usage information required resource assignment is flexible changes based on task's usage and load	6.2.3.1
Scheduling Order	according to priority class across classes according to arrival time in class evicted tasks rescheduled based on their first arrival time of the system	6.2.3.2
Preemption	priority evictions : a high priority task can preempt the execution of a lower priority class task	6.2.3.3
	memory evictions : tasks can be evicted if server's memory capacity is exceeded	6.2.3.4

source inefficiencies due to overestimated resource requirements of tasks [3, 50]. Motivated by the resource inefficiency and overbooking problem, we employ a slot-based task assignment which does not require user-provided task resource requirements. With this approach, each server is configured with specific number of slots and a task is only dispatched to a server if there is an available slot. Slots act as tokens to limit the resource sharing on a server among multiple tasks. Hence, each task occupies exactly one slot independent of its resource usage or priority. Once a task is assigned to a slot, it immediately starts its execution and the server resources are dynamically allocated to it.

Slot-based resource assignment relies on actual resource usage of tasks. On one hand, slot-based resource assignment prevents resource waste due to overestimation of resources. On the other hand, it also restricts over-consolidation of tasks by limiting the number of tasks assigned to a server which may result in significantly increased

task response times.

6.2.3.2. Priority Scheduling. We model a discrete time scheduler which updates the task states and resources at each time slot, i.e., one second. At the beginning of each time slot, the scheduler attempts to schedule every queued task, including new arrivals, waiting tasks and evicted tasks, one by one until the queue is empty or all slots are occupied. If there are available slots in the system, the scheduler dispatches the task from the head of the queue and assigns it to an available slot randomly. However, when the system is full and a high priority task arrives, a task in execution with the lowest priority is evicted as shown in Figure 6.3. The evicted task is selected according to the eviction policy employed as described in Section 6.2.3.3. Upon the eviction of low priority task, the high priority evictor task immediately scheduled to the freed slot and starts its execution, while the evicted low priority task rejoins the central queue. To minimize evictions due to high priority arrival, the central queue is always kept sorted according to the priority class. Moreover, tasks in the same priority class are sorted according to their arrival time to the system. Note that, the arrival time of an evicted task which rejoins the queue still refers to the time stamp when the task first arrived at the system. This insures that evicted tasks are the first ones to be scheduled, once slots are available again.

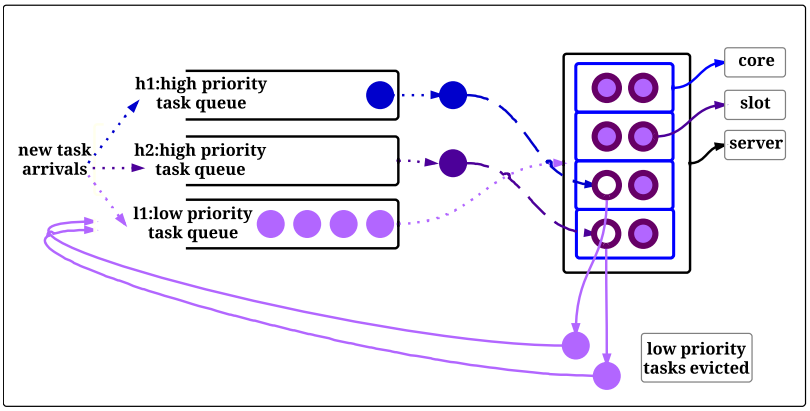


Figure 6.3. Priority scheduler design.

Slot-based resource assignment ensures that a task can not occupy more than one slot and also cannot run on more than one core. At every time slot, the maximum CPU capacity allocates to a task is limited by the slots and by the core capacity, which is

assumed to be normalized CPU capacity divided by the number of cores. The details of multicore, slot system and the computation of task execution times are shown in the Section 6.2.4.

6.2.3.3. Priority Eviction Model. Heterogeneous workloads have diversified requirements, i.e, not all of them have equal importance or priority, or equally sensitive to response times. In order to provide class specific service level objectives, current priority schedulers use evictions. This leads us to explore priority scheduling with various eviction policies in the following. In particular, we consider preemptive scheduling, i.e., a high priority task can preempt the execution of any task from a lower priority class creating a performance trade-off between different priority classes. More in detail, priority eviction is triggered by a high priority task arrival to a fully occupied system, i.e, when there is no available slot in the system.

We investigate the system under two resuming schemes: resume and non-resume, where the execution state of the task is not saved at the time of eviction for the latter which causes wasted executions. In non-resume systems, the task loses its execution at the time of eviction and starts from the beginning when restarted. Resume systems might seem more performant, but in practice not all applications can support it and/or the handling of the task state might overcome the benefit of continuing the execution of the task. For example, the states of tasks are not saved when evicted in Google cluster [29]. More in general, big clusters are usually non-resume systems since saving the task state at the time of eviction is too costly in terms of time and space. Due to explained reasons, in this study we mainly focus on non-resume systems.

Different eviction policies result in different eviction costs making the eviction policy a key parameter. An evicted task is always selected from the lowest priority class in execution from the whole cluster and the evictor task is from a strictly higher priority class. Since the scheduler works according to slot-based resource assignment, exactly one task is evicted for each evictor task. If there is only one such lowest priority task that task is evicted. If multiple lowest priority tasks exist in the cluster,

the eviction policy decides among them which one to evict according to the policies described below:

- The random eviction policy **RND** ignores executions of tasks and randomly selects the evicted task from the lowest priority class.
- The most recently started policy **MRS** takes into account both the priority class and start time of the task and selects the most recently started task to be evicted which is from the lowest priority class.
- The least recently started policy **LRS** is similar to MRS, where LRS select the least recently started task from the lowest priority class.

We propose to evaluate these three policies in order to capture the effect of eviction policy. The intuition behind MRS is to minimize the wasted execution by always selecting the youngest task. On the other hand, we implement LRS to evict the task which executed longer presumably allocating more resources. Lastly, we introduce RND to get a mixture of LRS and MRS also adding randomness to the system.

6.2.3.4. Memory Exceed. Unlike CPU usage, memory usage can not be adjusted according to the available memory capacity, i.e,. a task can continue its execution when its given CPU rate is reduced just runs slower. However, a task can not continue its execution with unsatisfied memory requirement [97]. When a task requires more memory than the available memory, it is evicted due to memory exhaustion. In contrary to priority evictions, memory evictions can be experienced by every priority class including the highest priority class. While priority evictions are triggered by the arrival of a high priority task to a fully utilized system, memory evictions occur when the server memory capacity is exceeded. In non-resume systems, both priority and memory evictions cause the loss of the tasks execution at the time of eviction.

The Google trace provides sampled task memory usage values for every 5 minutes period, however, majority of the tasks execute less than 5 minutes. Hence, we do not have enough information to model how the memory usage changes over time.

Without having this information, we assume that a task starts with a lower memory usage than its average usage, and the memory usage of the task grows over time [97]. For tasks having multiple sampled memory usage values, i.e,. execute more than 5 minutes, we take the average of sampled values. More in detail, we model the memory exhaustion and consequent memory evictions as follows. Irrespective of the currently available memory and memory usage of the task, the task is immediately assigned to an available slot. If available memory in the assigned server is greater than or equal to the tasks average memory usage, memory is allocated according to the tasks usage. If the available memory is insufficient, the scheduler allocates all the currently available memory to the task and records a checkpoint (X) as shown in Figure 6.4.

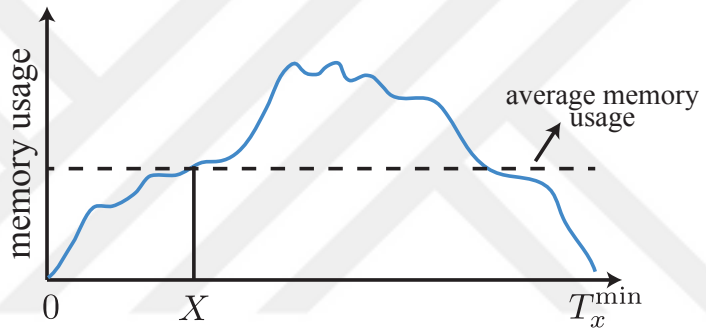


Figure 6.4. Memory eviction model: X denotes the check point to be compared against the estimated minimum execution time of the task, T_x^{min}.

The checkpoint is randomly selected between [0 T_x^{min}] where T_x^{min} is the estimated minimum execution time of the task. T_x^{min} is computed simply by dividing the CPU demand of the task with the largest core capacity. The checkpoint indicates that this task is going to require more memory than the currently allocated one. Up to the checkpoint the task has tolerance to run with the provided memory, however after the checkpoint it requires extra memory, i.e., reaches to the average memory usage provided by the trace. Once the checkpoint is reached, the scheduler checks again, whether the memory usage of the task can be satisfied or not with current available memory. If so, the extra needed memory is allocated to the task and the task continues to execute. Otherwise memory limit is exceeded and the scheduler takes action and evicts one or more tasks based on the memory eviction policy in place.

The eviction policy decides which task to evict among the ones running on the

memory exhausted server. We propose two memory eviction policies:

- LSF: based on time, the last started task with unsatisfied memory requirements is evicted first, independent of its priority.
- LPF: based on priority, the lowest priority tasks are evicted first, starting from the most recently scheduled one until the memory requirement of the high priority task is satisfied. With this policy, more than one task can be evicted in order to provide enough memory for a high priority task.

With these memory eviction policies, we investigate the tradeoff between evicting multiple low priority tasks and evicting the task that requires more memory. Essentially, LPF eviction policy uses prioritization while selecting the task to evict and allows to evict more than one task at a time in order to satisfy the memory requirements of a high priority task. However, evicting multiple tasks might be too costly in terms of wasted executions. As a result, we implement LSF which limits the number of evictions to prevent the system rescheduling overhead.

6.2.4. Response Time Model

Response time of a task, R is composed of two terms: T_x is the successful execution time of the task and the rest, $T_w = R - T_x$, is the time spent other than useful computation. Although some tasks can depart without experiencing evictions, it is possible for a task to get evicted repeatedly. From arrival to departure, the task make transitions between several states which are shown in Figure 6.5, where concrete lines shows mandatory costs states and actions and dashed ones are optional.

6.2.4.1. Successful Execution. The first contributor of R is the execution time of the task (T_x), which is obtained when the CPU demand of the task (Δ_C) is satisfied by the integral of the assigned CPU rate (Γ_c) as $\Delta_C = \int_0^{T_x} \Gamma_c(t)$.

By using slot-based priority scheduling, concurrently running tasks equally share

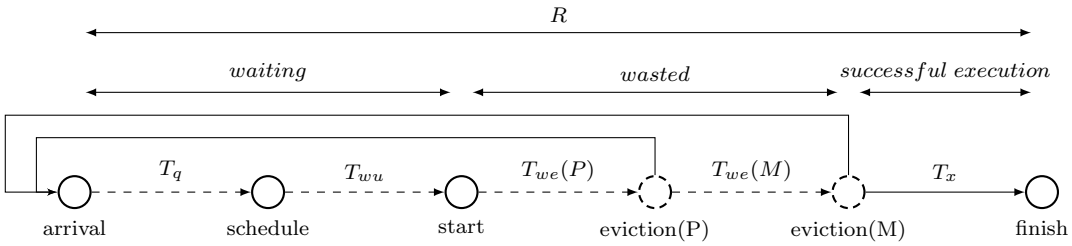


Figure 6.5. The breakdown structure of response time of a task.

the CPU capacity of the server (C). Moreover, tasks can not run on more than one core, and more than one slot. Hence the maximum CPU rate that a task can get is limited by the core capacity and the minimum CPU rate that a task can get is limited by the average CPU capacity per slot, $\frac{C}{N_s} \leq \Gamma_c(t) \leq \frac{C}{N_c}$. While the minimum T_x is determined by the number of number of cores (N_c), maximum T_x is limited by the number of slots (N_s) as shown in Equation 6.1.

$$\frac{\Delta_C N_c}{C} \leq T_x \leq \frac{\Delta_C N_s}{C} \tag{6.1}$$

We calculate the assigned CPU rate for a task at each time slot, by dividing the total CPU processing capacity among tasks running on the same server. With this approach, tasks can use spare CPU capacity to run faster. Since a task can not use more than one core, CPU rate is adjusted to one core capacity, if the number of running tasks is smaller than number of cores on that machine as shown in the Equation 6.2.

$$\Gamma_c(t) = \begin{cases} \frac{C}{N_r(t)} & \text{if } \frac{C}{N_r(t)} \leq \frac{C}{N_c} \\ \frac{C}{N_c} & \text{if } \frac{C}{N_r(t)} > \frac{C}{N_c} \end{cases} \tag{6.2}$$

where $\Gamma_c(t)$ is the assigned CPU rate of a task at time slot t. $N_r(t)$ is the number of concurrently running tasks on server s at time slot t. $\frac{C}{N_c}$ is the core capacity, which is independent of time and constitutes an upper bound for the CPU rate a task can get. The assigned CPU rate of a task is updated every time slot since it is dependent on the number of tasks co-executing.

6.2.4.2. Waiting Time. The second term of R, waiting time is composed of two terms T_{wu} which is the wake up time for a sleeping server, T_q is the time spend in the queue. While T_q is highly dependent on the system load and scheduling policy, T_{wu} is based on server properties. In our study we take $T_{wu} = 30$ *sec*.

6.2.4.3. Wasted Time. Last term T_{we} is the total wasted executions due to eviction of tasks in non-resume systems. In non-resume systems, task states are not saved hence tasks lose their execution at the time of eviction. When a task is evicted, it frees the allocated resources and rejoins to the central queue and waits for scheduling again. Therefore, T_{we} strictly depends on scheduling algorithms, system properties and system load. Evictions can take place due to memory exceed ($T_{we}(M)$) and high priority ($T_{we}(P)$) arrivals.

6.2.5. Dynamic Power Management

In our system, all servers are equipped with the technology supporting three operation modes, i.e., *on*, *idle*, and *sleep* as shown in Figure 6.6. The sleep states require explicit (de)activation commands which are subject to a dynamic power management policy.

In our system, we implement two policies. The first policy is *remain idle (IDL)* where IDL switches between *idle* and *on* states. Once there are no tasks running on that server and no tasks waiting, the server immediately enters idle mode and stays in idle mode until a task is assigned to it. Transitioning between *idle* and *on* states incurs neither delay nor power cost.

The second policy is *immediate sleep (SLP)*: once there are no tasks running on that server and no tasks waiting, the server immediately switches to sleep mode [98] in order to save energy. SLP policy does not include *idle* state since it directly activates sleep mode. This policy allows to save additional energy during low load periods, but incurs wake up delay which is $T_{sleep \rightarrow on} = 30s$. In addition, during wake up the

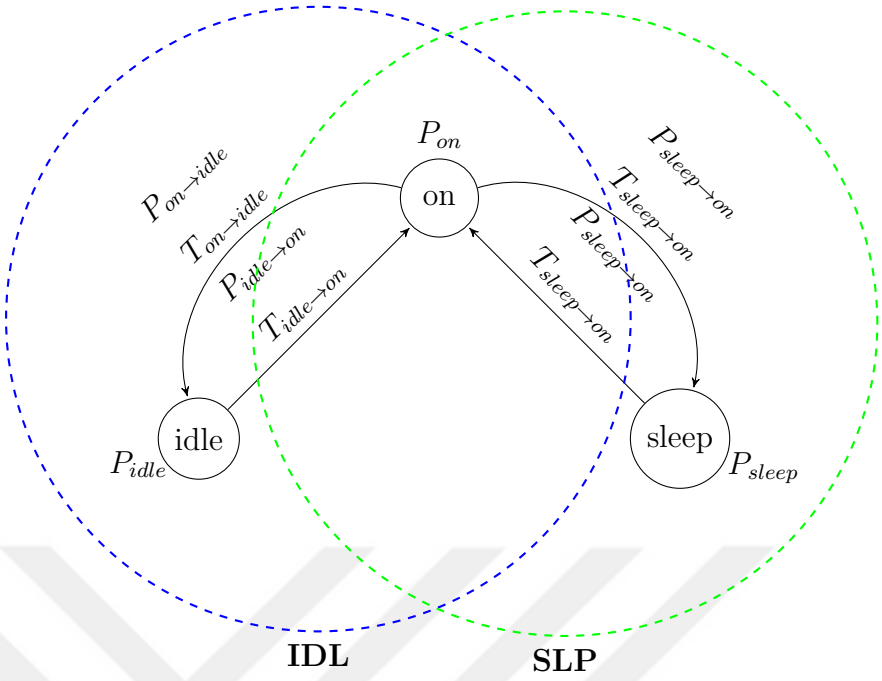


Figure 6.6. The state transitions of a server with IDL and SLP power management policies.

server operates on peak power consumption, $P_{sleep\to on} = P_{peak}$. In contrast, switching from *on*/*idle* to *sleep* bears no delay nor energy cost. We show the transitions of two dynamic power management policies in Figure 6.6. Since sleep and wake up decisions are triggered by the tasks in the system, the number of *on* servers naturally fluctuates with the workload.

6.2.6. Energy Consumption Model

We model the power consumption of servers, to calculate the energy consumed by the system. We consider sleep mode in order to save energy, rather than turning off servers due to long boot times (around 250 seconds) which is impractical in many cases [82]. We model the power consumption of an *on* server with a fixed term, i.e., power consumption at the *idle* state, and a load-dependent term that increases linearly with the memory and CPU usage [1, 10, 50] as follows:

$$P_{on} = P_{idle} + P_{cpu}U_{cpu} + P_{mem}U_{mem} \tag{6.3}$$

where P_{cpu} and P_{mem} are the peak power consumptions of CPU and memory, while U_{cpu} and U_{mem} denote the normalized utilization of CPU and memory, respectively. Hence, the peak power consumption of an *on* server equals to:

$$P_{peak} = P_{idle} + P_{cpu} + P_{mem} \tag{6.4}$$

6.3. Workload and Server Environment

Our framework allows both: (i) synthetically generated task and server attributes following some predefined distributions such as exponential or Pareto; and, (ii) input based on real system traces and characteristics. In this study, we focus on the second one to be closer to the real world conditions. In particular, we consider the publicly available Google Cluster trace, which represents a rich heterogeneous workload mix, on a large heterogeneous cluster. The trace provides information of both the workload and cluster environment. We present the properties of the workload in Chapter 5.

6.3.1. Server Heterogeneity

Today's data centers are usually composed of different server types equipped with different CPU and memory capacities, and characterized by different power consumption values. The Google Cluster trace provides normalized resource capacities for different server types. However, the power profiles of the servers are not provided. To overcome this limitation, we integrate the power usage breakdowns of the servers taking the reference values given in [76, 99]. The capacity and power characteristics of the four dominant server types are summarized in Table 3.1. Note that, we assume a linear relationship between the CPU/memory capacity and power consumption terms (P_{cpu}, P_{mem}) across the different server types.

6.4. Design Comparisons

To better understand the tradeoffs and correlations between different system design decisions, we conduct several analysis under different system configurations. First, we analyze the effect of priority and memory eviction policies separately, then we investigate their combined effect. The details of the experiment settings are described in Table 6.2. The first set is composed of three experiments ($\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$), and shows the effect of priority eviction policies. The second set of experiments $\mathcal{S}_4$ and $\mathcal{S}_5$ aims at studying the effect of memory eviction policies. Lastly, in the third set from $\mathcal{S}_6$ to $\mathcal{S}_{11}$, we analyze all combinations of memory eviction and priority eviction policies. With respect to the dynamic power management policies, we always use the SLP dynamic power management policy.

We evaluate a cluster of 125 heterogeneous server nodes, each containing 8 cores using our proposed simulation framework. The distribution of the server types and capacities reflect the four dominant types identified from the Google Cluster trace and presented in Section 6.3.1 and summarized in Table 3.1. The number of slots for each server is scaled according to the CPU capacity of the server. More precisely, A, B and C type servers are holding 24 slots while D type servers are holding 48 slots.

6.4.1. Performance Metrics

In non-resume systems, the wasted executions are non-negligible and have a significant negative impact on the response times as well. Due to restarts of evicted tasks, the utilization of non-resume systems increases drastically. Therefore, we evaluate each set of different priority and memory eviction policies from a three-fold point of view.

First, we consider the response time. Typically, the average response time is used as the primary performance metric for scheduling. However, the average response time is not a perfect metric to quantify the performance in multi-class systems with varying number of arrivals. Indeed, the average response time is priority-oblivious, hence it is highly correlated with the priority class having the largest population,

Table 6.2. System design decision options.

Objective	Setting	Priority Evictions	Memory Evictions
Effect of priority eviction policy with unlimited memory	$\mathcal{S}_1$	LRS	Unlimited memory
	$\mathcal{S}_2$	RND	Unlimited memory
	$\mathcal{S}_3$	MRS	Unlimited memory
Effect of memory evictions with no priority scheduling	$\mathcal{S}_4$	No priority	LPF
	$\mathcal{S}_5$	No priority	LSF
Effect of priority eviction policy and memory eviction policy combined	$\mathcal{S}_6$	LRS	LPF
	$\mathcal{S}_7$	LRS	LSF
	$\mathcal{S}_8$	RND	LPF
	$\mathcal{S}_9$	RND	LSF
	$\mathcal{S}_{10}$	MRS	LPF
	$\mathcal{S}_{11}$	MRS	LSF

which typically is a low priority class. When designing priority schedulers for highly heterogeneous workloads, on one hand, we want to meet the stringent SLOs of the high priority class tasks. On the other hand, we want to avoid significant performance degradations of low priority class tasks due to starvation. To better take into account these conflicting design goals in a unique number, we propose a new metric V to quantify the performance of a priority. V is the averaged weighted response time, where the weights are the priority classes and independent of the number of arrivals of classes. The new metric V is defined as:

$$V = \frac{\sum_{i \in K} wR(C_i)}{|K|} \tag{6.5}$$

where w is the weight vector, K is the set of priority classes. We take the weight vector proportional to the priorities, $w = K + a$, where $a > 2$ in order not to miss the effect of class 0. This metric has the advantage of being independent of the per class population size while taking into account the response times achieved by all classes.

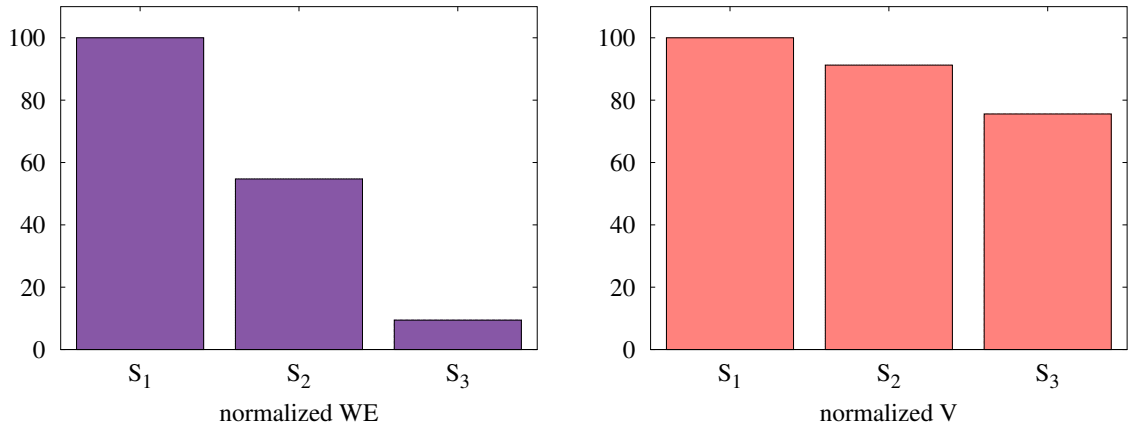
Second, we want to quantify the resource (in)efficiency caused by both priority and memory evictions. We use WE which is the accumulated wasted executions over all evictions, i.e., priority and memory evictions. WE is calculated as the amount of CPU time wasted, i.e,. time spend to execute with unit CPU capacity (1.0), hence it is in $cpu \cdot sec$.

Finally, we are interested in the total energy consumed by the cluster E, which is the integral of the power consumption values for all servers over time. In this study, we present the normalized V, WE and E with respect to the highest value.

6.4.2. Effect of Priority Eviction Policy

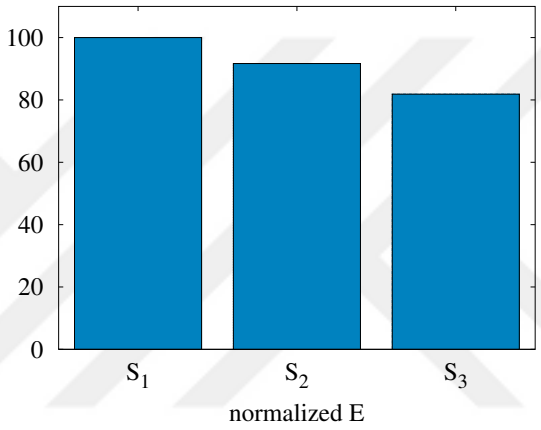
We start by evaluating the effect of different priority eviction policies on V, WE and E in Figure 6.7. Here, we avoid memory evictions by using an unlimited memory system and disregard the energy consumption of the memory. When evictions take place, the system utilization increases due to restarted executions of evicted tasks. The increased system load have a significant negative impact on response time and wasted resources. As a result, it is very important to minimize the impact of evictions, i.e., wasted resources in order to prevent system to enter a negative feedback loop.

Starting from Figure 6.7(a), one can observe that the MRS policy ($\mathcal{S}_3$) is able to reduce the wasted resources by up to 90% compared to LRS ($\mathcal{S}_1$) by selecting the youngest task as the one to be evicted. The significant reduction in wasted executions reduces also the overall system utilization having a positive effect also on V and E. MRS improves V by more than 20% compared to LRS and by 10% compared to RND ($\mathcal{S}_2$) (see Figure 6.7(b)). Response time of highest priority tasks are only affected by the arrival distribution and resource demand of their own class. Since class 0 is the lowest priority class, its response time is significantly effected by the system load. By selecting the youngest task to be evicted MRS helps to reduce the increase in system load due to evictions, hence improves low priority response time as well as high priority response time which are included in V. Similar results are also obtained for the consumed energy E (see Figure 6.7(c)). LRS policy consumes more energy by executing more total load.



(a) Wasted executions.

(b) Weighted response time.



(c) Energy.

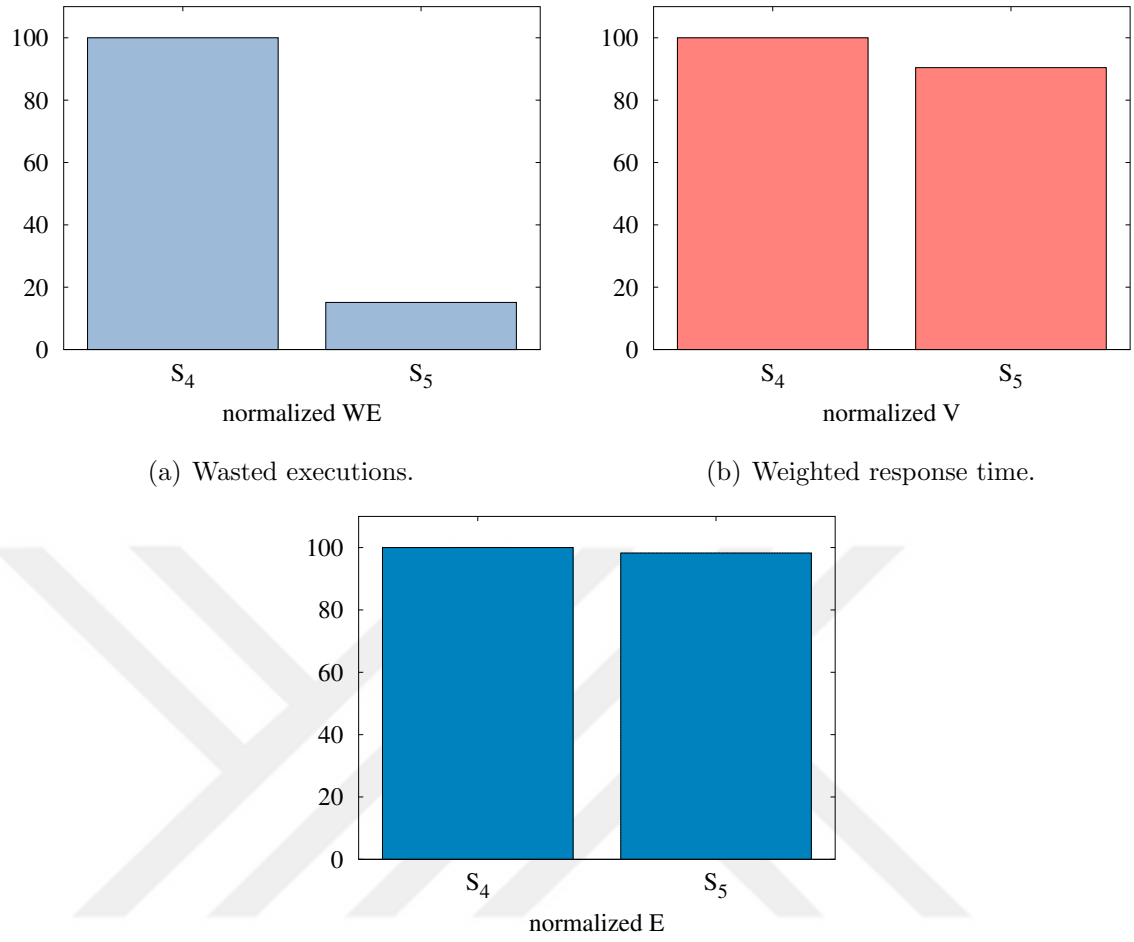
Figure 6.7. Effect of priority eviction policies on WE, V and E with unlimited memory.

Overall, for all three performance metrics (V, WE and E), MRS ($\mathcal{S}_3$) gives the best performance.

6.4.3. Effect of Memory Eviction Policy

Next, in Figure 6.8, we investigate the effect of memory eviction in the absence of priority scheduling to eliminate the effect of priority evictions, i.e., task are scheduled in first come first serve fashion. We compare the LPF ($\mathcal{S}_4$) and the LSF ($\mathcal{S}_5$) memory eviction policy.

LPF is allowed to evict multiple tasks in order to satisfy the memory constraint of a high priority task. Consequently, the wasted executions with LPF is significantly



(a) Wasted executions.

(b) Weighted response time.

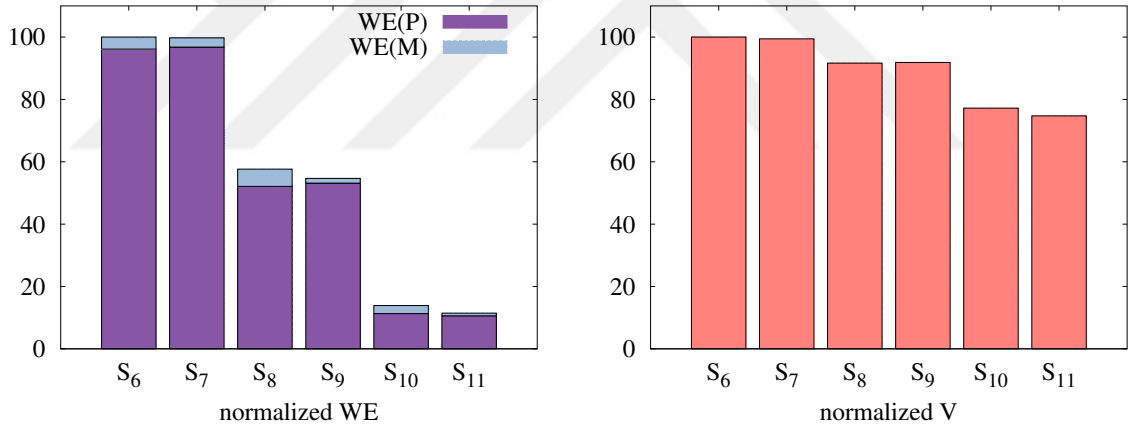
(c) Energy.

Figure 6.8. Effect of memory evictions with no-priority scheduling on WE, V and E.

higher than with LSF as shown in Figure 6.8(a). This result shows that evicting multiple tasks in order to fully satisfy a higher priority task's memory usage does not compensate the negative impact of increased system load due to restarts of evicted tasks. However, in terms of V and E the gap is not that significant compared to WE. The reason behind is reschedulings due to memory evictions are rare compared to other scheduling events, i.e., schedulings of new arrived tasks. Hence, the effect of memory evictions on V and E are not severe. V and E are dominated by normal executions. Therefore, we do not observe significant improvement with LSF compared to LPF, due to small degree of effect of memory evictions on V and E. Nevertheless, LSF provides a 10% improvement on both V and E by evicting only the last scheduled task.

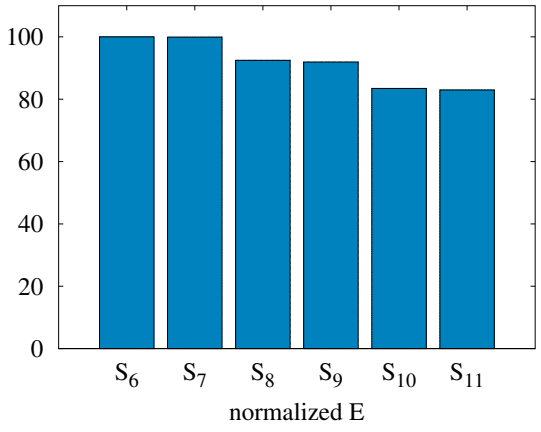
6.4.4. Effect of Priority and Memory Eviction Policies

Lastly, in Figure 6.9, we analyze all the combinations of the previously presented priority and memory eviction policies, i.e., from $\mathcal{S}_6$ to $\mathcal{S}_{11}$. One can observe from Figure 6.9(a) that the priority eviction policy dominates the wasted executions. The reason is that priority evictions are more frequent than memory evictions. Additionally, LPF memory eviction policy results in more wasted executions compared to same priority eviction policy in combination with LSF. The difference between LPF and LSF is more clear with $\mathcal{S}_8$ to $\mathcal{S}_9$. Consequently, as expected, MRS achieves the best results with more than 80% improvement on WE compared to LRS. Moreover, within the same priority eviction policy, the LSF memory eviction policy causes less wasted executions similar with our previous results, i.e., one does not observe any detrimental effect by combining the priority and memory eviction policies.



(a) Wasted executions.

(b) Weighted response time.



(c) Energy.

Figure 6.9. Effect of priority eviction and memory eviction policies on WE, V and E.

In terms of V and E, $\mathcal{S}_{11}$ provides the best results which is the combination of MRS and LSF policies. More in detail, V can be improved by more than 20% and E can be reduced by almost 20%. Although the major part of the improvement is obtained from the priority eviction policy, i.e,. MRS, LSF also improves both V and E with all priority policies compared to LPF. Surprisingly, these experiments shows us, it is possible to execute the same workload with better response time (see Figure 6.9(b)), less wasted resources (see Figure 6.9(a)) and smaller energy budget (see Figure 6.9(c)) at the same time by using right policies ($\mathcal{S}_{11}$).

Overall, evictions have nonnegligible negative impact on system performance, while priority evictions are more frequent than memory evictions as observed in Google cluster trace. Since the system utilization increases due to restarted executions of evicted tasks, different eviction policies results in varying wasted executions which effects the system load and performance. MRS priority eviction policy in combination with LSF memory eviction policy helps to decrease the wasted resources by 80% and improves V and E by 20% compared to LRS, LPF combination.

6.5. Chapter Summary

Motivated by high complexity of system-workload and significant resource inefficiency in priority based schedulers, we propose a trace-driven cluster management framework that enables exploring the design space of scheduling policies with a particular focus on the impact of task evictions. To explore various what-if scenarios of diverse settings of priority scheduling, we develop a general detailed system model that captures the execution progress and response times when executing complex workloads on heterogeneous systems. Driving our framework with Google Cluster production traces, we evaluate the tradeoff between resource inefficiency and task response times of different priorities under different combinations of scheduling policies and system configurations.

7. PRIORITY SCHEDULING FOR HETEROGENEOUS WORKLOADS

Large computing clusters are executing highly heterogeneous bursty workloads with different priorities, resource demands and performance objectives nowadays. As the schedulers control the class specific performance by evictions, a noteworthy amount of the computing resources are wasted by the lost executions of the evicted tasks. This study presents a system model and a slot-based priority scheduler that captures the execution progress, evictions and response times when executing complex workloads on heterogeneous systems. To better understand the impact of evictions, we first analyze simple eviction policies, and wasted resources associated with evictions by using trace-driven simulation of Google cluster trace. Our key finding is that, evictions severely increase system load which lead to a significant increase on response times, especially of low priority classes, due to *repetitive* evictions. Furthermore, we investigate and verify the reasons of repetitive evictions.

7.1. Motivation

Energy consumption of data centers presents a significant cost of operating cloud services. While data centers constitute the primary energy consumers of ICT, a significant amount of energy is wasted due to non-optimized resource scheduling and inefficient designs [100]. Hence, there is an increasing amount of interest in improving the energy efficiency of data centers [101]. Vast amount of work focus on reducing the energy consumption of unused resources by dynamic capacity provisioning, and adjusting the active number of servers according to the arrivals [8,11]. However, heterogeneity [3] is usually overlooked and turns into a major challenge for designing schedulers.

Cloud data centers usually receive a large number of heterogeneous resource requests with highly varying resource demand, priorities and performance objectives. In addition to the workload heterogeneity, large clusters usually consist of heteroge-

neous machines in terms of processing, storage and memory capacity, and number of cores. Providing class specific response times on heterogeneous clusters is highly challenging, due to high variability in the resource demand of classes, i.e., some tasks take much longer or require more resources than other tasks [3]. Proposed solutions that try to minimize the negative impact of high variability and avoid the degradation of tail response times [4] usually rely on user defined resource allocations [3], speculative executions [70], and evictions (terminating and restarting of tasks) [69] to mitigate stragglers. However, these approaches lead to significant waste of computing resources. While dynamic capacity provisioning schemes focus on the energy consumption of unused resources, there exists a nonnegligible degree of energy and resource inefficiency on the usage of computational resources, mainly caused by overbooking and lost executions of unsuccessful events which are usually overlooked.

A goal of this study is to address *resource inefficiency* and we develop a comprehensive system model that captures the major issues of inefficient resource usage and lost executions associated with the evictions. We propose to apply slot-based scheduling to overcome overbooking problem that arises from the user defined resource reservations, which are usually overestimated [3]. Slot-based scheduling not only eliminates the need of using tasks' resource information but also limit the number of co-executing tasks on a server. Our proposed system model also enables us to analyze several priority scheduling, eviction polices, system settings and slot configurations to reveal insights for better understanding the impact of evictions and eviction policies. We propose to apply three eviction policies, namely MRS, RND and LRS to the slot-based priority scheduler and evaluate on Google cluster trace [33], which is a mix of different workloads. We find out that evictions have a significant negative impact on low priority classes especially with bursty workloads. Hence, selecting right eviction policy is important to minimize the negative effect of evictions. Surprisingly, we find out that RND and LRS, tend to cause more repetitively evicted tasks than MRS which results in outlier response times. Furthermore, we investigate and verify the reasons of repetitive evictions.

7.2. System Model

To explore various what-if scenarios of diverse settings of priority scheduling, we develop a general detailed system model that captures the execution progress and response times when executing complex workloads on heterogeneous systems. The system model mainly consists of three parts: the central queue, heterogeneous servers, and the scheduler. The tasks arriving to the system enqueue at the central queue and wait for scheduling. The queue is kept sorted according to the priority class and then according to the arrival time. The scheduler dispatches the task from the head of the queue and schedules the task based on the server availability and scheduling discipline employed. The details of the developed system model is presented in Chapter 6.

Table 7.1. Server configurations.

	Ratio	CPU	Memory	Quantity	Cores	Slots
A	54%	0.5	0.50	69	8	16
B	31%	0.5	0.25	38	8	16
C	8%	0.5	0.75	10	8	16
D	7%	1.0	1.00	8	8	32

The dominant 4 types of server are shown in Table 7.1 which we use in this study. In particular, we evaluate a data center of 125 heterogeneous multicore servers. For the analysis in this study, we simulate the system for 15 hours with 125 heterogeneous servers which are subject to a sleeping policy, i.e., machines immediately enter sleep mode once there are no tasks running and take 30 seconds to wake up. The server configurations are shown in Table 7.1, and the number of finished tasks is around 68.000.

7.3. Priority Scheduling Analysis

Not all tasks have equal importance or priority, or equally sensitive to response times. Current priority schedulers adopt eviction policies to provide class specific ser-

vice level objectives. This leads us to explore priority scheduling with various eviction policies in the following. We investigate the system under two resuming schemes, resume and non-resume, where the execution state of the task is not saved at the time of eviction for the latter which causes wasted executions (WE). WE is calculated as the amount of CPU time wasted, i.e,. time spend to execute with unit CPU capacity (1.0), hence it is in $cpu \cdot sec$. Although resume systems are well investigated, non-resume systems are little known due to their complex analytical tractability and instability conditions.

The aim of our analysis is three-fold. First, we show how the wasted resources and response times are effected by the introduction of priority scheduling, under resume and non-resume schemes. Second, we look deeper into the eviction policies, we investigate the reasons and underlying phenomenon of repetitive evictions. Third, we apply thresholding to minimize the wasted executions and improve the response times increased by repetitive evictions of the low priorities. For the analysis in this section, we do not apply memory limits in order to isolate the effect of priority evictions.

7.3.1. Preemptive Priority Scheduling

In particular, we consider preemptive priority scheduling, i.e., high priority tasks can preempt any execution of low priority tasks. Upon preemption, low priority tasks immediately rejoin the central queue. The formal definitions of eviction events and scheduling order are described in the following.

Eviction event: A task from class l can only be evicted due to priority when a task from class h arrives where $l < h$ and there is no available slot in the system. Hence, priority eviction only occurs when the system is fully utilized and there is at least one lower priority task running when high priority arrives to the system.

Scheduling order: The central queue is sorted according to priority class where highest priority class is at the head of the queue. In each priority class, the tasks are sorted according to arrival time to the system in ascending order. So the scheduling

works in first come first serve (FCFS) order in each priority class. We note that, the arrival time of an evicted task, refers to the time stamp when the task first arrives at the system.

Exactly one task is evicted for each evictor task since the scheduler is slot-based. If multiple lowest priority tasks are running in the cluster, the eviction policy decides which one to evict according to the proposed eviction polices, i.e,. MRS, RND, LRS, which are introduced in Chapter 6.

7.3.2. Impact of Priority Evictions

In practice not all applications can support saving the task state and/or checkpointing of the task state might overcome the benefit of continuing the execution of the task [29]. More in general big clusters are usually non-resume systems since saving the state of the evicted task is too costly in terms of time and space. We compare resume and non-resume systems in terms of average response time, wasted executions and response time of lowest and highest priority class with MRS, RND and LRS eviction policies in Table 7.3.2. We also show the average response time of evicted tasks and non-evicted tasks from the class 0, $R(C_0)_{evicted}$ and $R(C_0)_{non-evicted}$ respectively in order to show the effect of evictions on tasks other than classes.

In resume systems, since evicted tasks continue their execution from the interruption point, wasted executions due to evictions are not experienced. Hence the average response time of MRS, RND and LRS policies under resume scheme is similar since in resume systems only the order of executions changes with the evictions. When we look into the evicted tasks, $R(C_0)_{evicted}$ shows an increase of 20% from MRS to LRS, where $R(C_0)_{evicted}$ highly depends on the repetitive evictions which is decided by the eviction policy. On the other hand, in non-resume systems the wasted resources have a significant negative impact on response time.

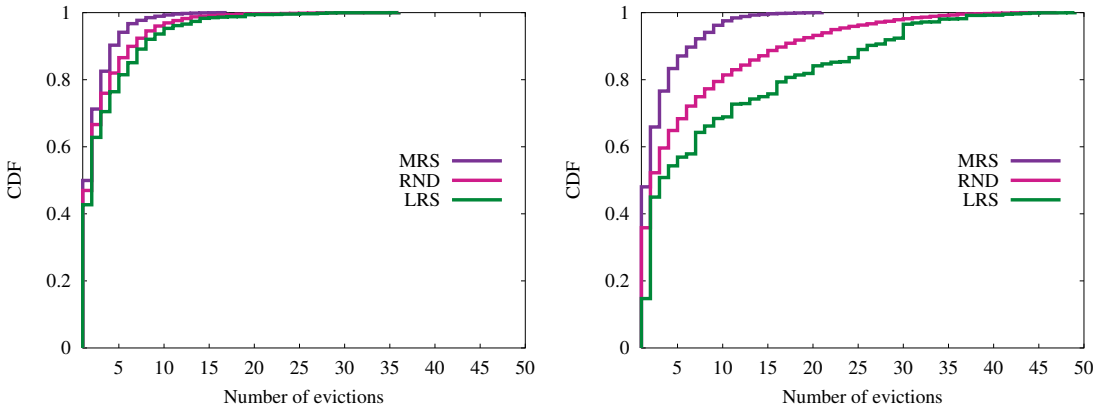
In non-resume systems, the utilization increases due to restarted executions of evicted tasks as shown in Table 7.3.2, the average response time is almost doubled

Table 7.2. WE and R with MRS, RND and LRS eviction policies under resume and non-resume systems.

	Resume			Non-Resume		
Metric	MRS	RND	LRS	MRS	RND	LRS
WE(10^9)[cpu.sec]	0	0	0	219	878	1014
R [sec]	1305	1286	1303	1779	3724	4750
$R(C_0)$ [sec]	2501	2490	2442	3959	10222	13678
$R(C_9)$ [sec]	2668	2652	2699	2877	3109	3125
$R(C_0)_{non-evicted}$ [sec]	1585	1540	1536	2616	8165	12302
$R(C_0)_{evicted}$ [sec]	4323	5200	5662	6877	17590	22218

compared to resume systems. MRS policy helps to decrease the wasted resources 70% and R is improved by 60% compared to LRS, by selecting the youngest task as the evicted one, but still non-resume system has higher response time compared to resume system. Response time of highest priority tasks are only affected by the arrival distribution and resource demand of their own class. Since class 0 is the lowest priority class, its response time is significantly effected by the policies employed. However, when we compare the response time of evicted and non-evicted tasks of class 0, we observe that the evicted tasks experience longer response with RND and LRS than MRS for both resume and non-resume schemes. For the resume case, the average response time of non-evicted class 0 tasks are similar with all eviction policies, but the response time of evicted tasks with RND and LRS are 17% and 23% higher than the response time of evicted tasks with MRS respectively. For the non-resume case the impact is increased by the increased wasted executions for both evicted and non-evicted task. Overall, different eviction policies result in varying wasted executions which affects the system load and performance, in non-resume systems.

Moreover, it is also important to observe how the evictions are distributed across tasks in the same priority classes, i.e., some policies tend to punish the same set of tasks by repetitively evicting them. Hence, eviction policy selection also affects the



(a) Resume. (b) Non-resume.

Figure 7.1. Distribution of number of evictions of evicted tasks with MRS, RND and LRS under resume and non-resume scheme.

probability of repetitive evictions. In Figure 7.1, we show the CDF of number of evictions for resume and non-resume systems. For both resume and non-resume systems, MRS is converging faster than RND and LRS. The differences between different policies become more visible for non-resume systems. Repetitive evictions result in extremely high response time of some tasks, obviously which is not desired. We can conclude that more repetitive evictions are observed with RND and LRS than MRS.

Surprisingly, a counter intuitive finding worth noting is that, RND and LRS policy produce stragglers by evicting the same set of tasks repetitively, hence it is more likely to observe outlier response times compared to the non-evicted tasks from the same class with RND and LRS. We find out that the eviction policy not only affects the extra load created by the evictions and resubmissions for non-resume systems, but also affects the probability of creating straggler tasks, by repetitively evicting same set of tasks. These findings lead us to discover the reasons of changing eviction behavior to better understand and optimize eviction procedure in the latter sections. Accordingly, motivated by the significant number of stragglers, we propose a simple thresholding mechanism to improve performance by getting rid of stragglers.

7.3.3. Understanding the Reasons under Repetitive Evictions

In this section, we turn our attention to understand the changing probability of repetitive evictions with MRS, RND and LRS. We start with the analysis of number of evictions, evicted tasks and maximum number of evictions with proposed eviction polices, then, we explain the details of our observations on repetitive evictions with formal definitions.

In Table 7.3, we analyze the evictions for LRS, RND and MRS policies under resume and non-resume systems. It is worth noting to point the distinction between *eviction* and *evicted task*. When a high priority comes to the system and can not find an available slot, then a slot is freed by preempting a low priority task, this event is called eviction. Hence, the task whom experience eviction is called the evicted task. A task can experience eviction more than once. Under resume scheme, even though, all policies experience similar number of evictions (NE), the number of evicted tasks (NET) are far from each other.

MRS evicts 25% more tasks than RND and 50% more tasks than LRS. At first look we expect MRS to pick the same set of tasks since it always selects the most recently started task to be evicted, which results in lower number of evicted tasks, but this is not the case. For non-resume systems, MRS experiences the smallest number of evictions while LRS experiencing the largest number of evictions. Therefore, maximum number of evictions ($\max(e)$) and average number of evictions per evicted task ($\overline{e}$) under non-resume system is higher than the resume system, due to increased system load. However, MRS still has the highest number of evicted tasks compared to RND and LRS.

In order to better understand the underlying reasons of unexpected eviction behavior, we make analysis with different policies, specifically in terms of NE and NET under both resume and non-resume systems. Since MRS and LRS are extreme eviction policies, we expect RND policy to be in between MRS and LRS, by introducing randomness, for different performance metrics.

Table 7.3. Analysis of eviction behavior for MRS, RND and LRS under resume and non-resume systems.

	Resume			Non-Resume		
Metric	MRS	RND	LRS	MRS	RND	LRS
NE	16022	16069	15970	18781	28367	28398
NET	7229	5663	4807	6850	4806	3152
$\overline{e}$	2.2	2.8	3.3	2.7	5.9	9.0
$\max(e)$	17	36	36	21	47	49

Statement 7.1. *For each priority class in the queue, evicted tasks are scheduled before the non-evicted ones from the same class.*

Explanation. When a task newly arrives to the system, it is added to the corresponding class queue and class queue is sorted based on the arrival time to the system. When a task is evicted, it joins back to the class queue to be scheduled again. Evicted tasks to be queued always have earlier arrival time to the system than non-evicted queued ones, hence they enter in front of the non-evicted queued tasks. This behavior relies on the scheduling discipline but it is independent of the eviction policy.

Statement 7.2. *In both resume and non-resume systems,*

$$NET_{MRS} \geq NET_{RND} \geq NET_{LRS} \tag{7.1}$$

When there is at least one already evicted task and one non-evicted task from the same class (which is the lowest class running) are running in the system, LRS selects the task from the set of already evicted ones. On the other hand, MRS always selects from the non-evicted ones. Hence, MRS tends to evict more unique tasks compared to LRS, i.e,. LRS tends to create more repetitively evicted tasks. RND swings in between MRS and LRS due to random selection.

Explanation. We use resume system to alleviate the effect of evictions on the system load. Different eviction behavior only appear when at least one already evicted task and one non-evicted task are running on the system from the lowest priority class. Since LRS selects the least recently started task (earliest start time), the evicted task is an already evicted task if at least one evicted task is running. Therefore the probability that the selected task is not evicted before is lower for LRS than MRS.

Statement 7.3. *While MRS always selects a non-evicted task, LRS always selects an evicted task.*

Explanation. Given that the system is fully utilized, when a high priority task arrives. The set of non-evicted tasks from the lowest priority class is denoted by N and the set of evicted tasks from the lowest priority class are running in the system is denoted by E, where $|N| = n$, $|E| = e$ and the selected task for eviction is denoted by η.

$$P(\eta \in N)_{MRS} = 1 \tag{7.2}$$

$$P(\eta \in E)_{LRS} = 1 \tag{7.3}$$

$$P(\eta \in E)_{RND} = \frac{e}{n+e} \tag{7.4}$$

with the exception of tasks that are started at the same time since MRS and LRS only use priority and start time information.

Statement 7.4. *In non-resume systems,*

$$NE_{MRS} \leq NE_{RND} \leq NE_{LRS} \tag{7.5}$$

Explanation. The underlying reason of this relationship is the increased system load due to larger amount of WE with RND. With RND policy, the system is more utilized

due to more WE compared to MRS.

$$NE = P(U = 100|arr^h) \times T \tag{7.6}$$

$$NE = \frac{P(arr^h|U = 100)P(U = 100)}{P(arr^h)} \times T \tag{7.7}$$

$$NE \propto P(U = 100) \tag{7.8}$$

where $P(U = 100|arr^h)$ is the probability of the system is fully utilized when there is a high priority arrival. Since RND policy is not designed to minimize the execution lost, wasted execution with RND is larger than MRS. As a result, tasks stays longer in the system with RND which increases the system load.

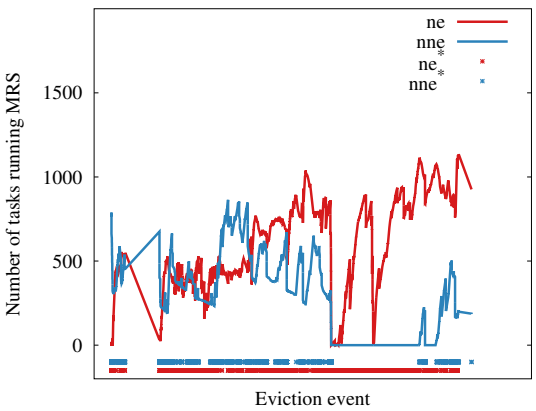
$$WE_{RND} \geq WE_{MRS} \tag{7.9}$$

$$T(U = 100)_{RND} \geq T(U = 100)_{MRS} \tag{7.10}$$

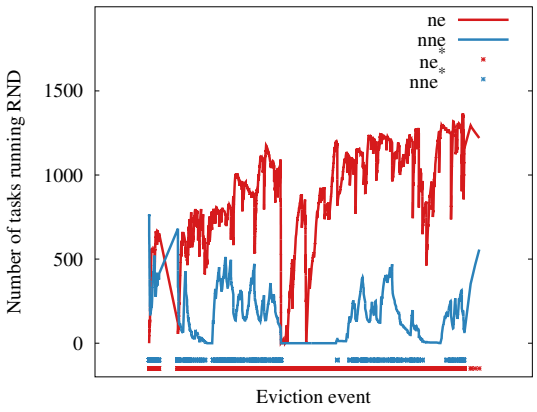
$$NE_{RND} \geq NE_{MRS} \tag{7.11}$$

where $T(U = 100)$ is the amount of the time that the system is fully utilized. Same explanations between RND and LRS. In order to verify this explanation, we compare RND, MRS, and LRS where evictions occur but WE = 0, i.e., under resume system. In resume systems, the eviction policy only changes the order of the schedule, but the evicted tasks conserve their state at the time of eviction and continue from the point of eviction when they are rescheduled. With same set up and input, in a resume system, we get same number of evictions $NE_{MRS} \approx NE_{RND} \approx NE_{LRS}$. We conclude that the reason behind larger number of evictions is the increased system load due to higher WE.

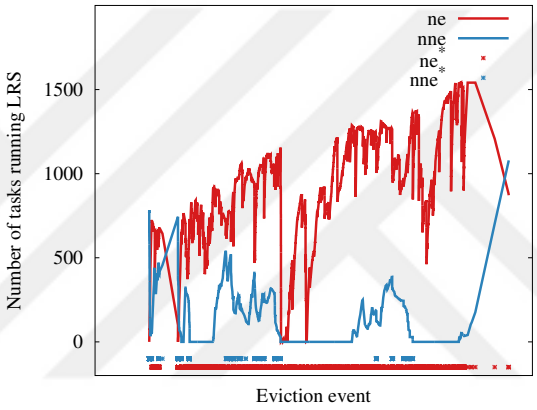
Evicted tasks are scheduled before the non-evicted ones from the same class since the scheduling order is class-based FCFS. Therefore, when the system becomes loaded and evictions take place, we expect ne to increase while nne is decreasing, where ne is the number of evicted and nne is the number of non-evicted tasks at the time of eviction.



(a) System state (ne, nee) at the time of eviction with MRS.



(b) System state (ne, nee) at the time of eviction with RND.



(c) System state (ne, nee) at the time of eviction with LRS.

Figure 7.2. Number of evicted and non-evicted tasks of class 0 running at the time of eviction and the evicted task with MRS, RND and LRS, where ne is the number of evicted and nne is the number of non-evicted tasks at the time of eviction.

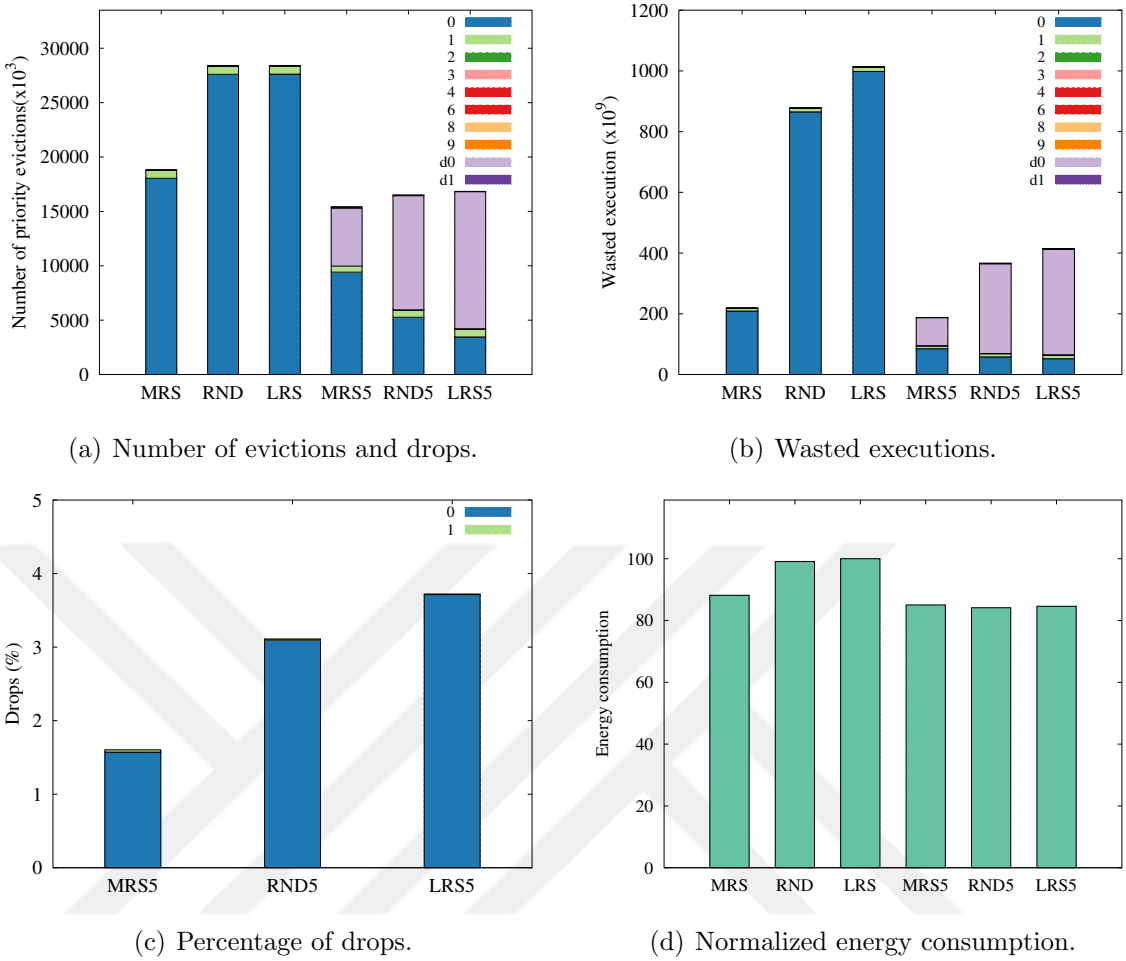
7.3.3.1. Simulation Verification. We further verify our statements, by inspecting simulation results, particularly by the system state of eviction. The *system state* consists of nne and ne tasks from the lowest class (class 0) running in the system when an eviction occurs. Figure 7.2 shows, the number of evicted and non-evicted tasks and the selected task for eviction at the time of eviction for MRS, RND and LRS, where each point indicates an eviction event. We can easily see that the tendency to select from evicted tasks increases in the order of MRS, RND and LRS, which supports our finding in Statement 7.2. While the lines show the number of evicted (ne) and non-evicted (nne) tasks at the time of eviction, the evicted task, whether it is an already evicted (ne^*) or non-evicted (nne^*) task, is shown in the bottom of the figure. Note

that, *ne* and *nne* include of only class 0 tasks, but there may exists evictions of other priorities which are not shown. The first observation is that, number of evicted tasks running in the system increases faster with RND and LRS than MRS. We see that LRS is converging too fast that the evicted tasks start to dominate the system immediately. Although the total number of eviction events with MRS is less than with RND and LRS, we can easily see that MRS selects much more non-evicted task to evict. In addition to the tendency to evict the same set of tasks, LRS experiences more evictions than MRS and RND due higher system load. Overall, we observe less NE and WE with MRS compared to RND and LRS, also MRS evicts from a more diverse task set. While designing priority schedulers, not only the selection of eviction policy but also its interaction with the scheduling order determines the system behavior.

7.3.4. Limiting Eviction

From our analysis, we observe that the scale of the negative effect of evictions changes with the eviction policy employed. Also the distribution of evictions among the tasks changes significantly. Task dropping can be applied to consecutive task failures which occur due to software crashes of a task [3]. Rescheduling of repetitively failing tasks creates stragglers in the system. In that case, thresholding can be useful to identify and eliminate these tasks from the system. Motivated by the high number of repetitive evictions on low priority tasks and the resulting waste of resources, we propose to apply thresholds on the number of evictions a task can experience and introduce task dropping in a non-resume system. The objective here is to use such a threshold to reach the optimal trade-off between the minimization of wasted resources and improvement of response time, particularly the response time of low priority tasks. We denote the eviction policy on which thresholds apply as MRSN, RNDN and LRSN where the threshold is N. We set the threshold as 5, to keep the percentage of drops under 5% and obtaining remarkable reduction in wasted executions, as a results of our experiments with different thresholds.

We investigate the number of evictions, wasted executions and percentage of drops by class summarized in Figure 7.3. We see that the eviction policies which allow drops,



(a) Number of evictions and drops.

(b) Wasted executions.

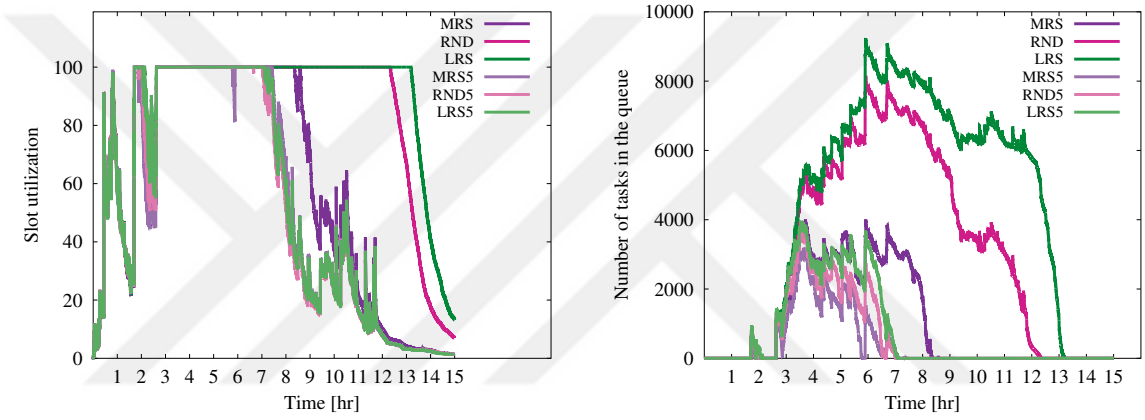
(c) Percentage of drops.

(d) Normalized energy consumption.

Figure 7.3. Number of evictions, drops, and wasted executions by priority class with MRS, RND, LRS and MRS5, RND5, LRS5.

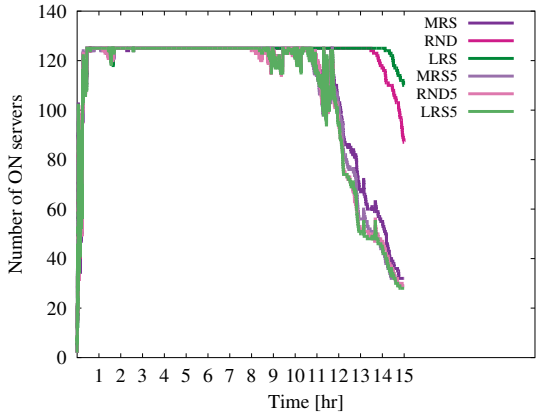
provide significant (>50%) reduction on wasted resources. Policies which allow drops force some tasks, which are repetitively evicted and restarted and exceed a certain number of evictions to leave the system and to mitigate the effect of straggling tasks. As a result, dropping is more effective on LRS compared to MRS since it experiences more repetitive evictions as shown in Figure 7.3(a) and in Figure 7.3(b). However, LRS5 drops two times more tasks compared to MRS5 as shown in Figure 7.3(c). When we compare LRS5, MRS5 and RND5, MRS5 shows the best improvement on wasted resources by combining smart eviction policy and dropping. As a result with MRS5, the breakdown of wasted executions mostly composed of just the lowest two priority class. Lastly, we evaluate the energy consumption of these policies in Figure 7.3(d). In terms of energy consumption, LRS has the highest energy consumption followed by the RND eviction policy since these policies generate more system load due to

resubmissions of evicted tasks. For the cases when dropping is not allowed, MRS reduces the energy consumption by 12% compared to LRS by reducing noteworthy amount wasted executions due to priority evictions. When dropping is allowed, we observe 15% and 16% energy reduction for RND5 and LRS5 respectively which is due to reduced system load by thresholding. Since MRS5 does not drop significant number of tasks, the energy consumption of MRS5 is reduced only by 4% compared to MRS. If drops are not allowed, MRS reduces evictions, wasted executions, energy consumption and hence system load significantly, by selecting the youngest tasks to evict, compared to both RND and LRS.



(a) Slot utilization.

(b) Number of tasks in the queue.



(c) Number of on servers.

Figure 7.4. Slot utilization, queue length and number of *ON* servers with MRS, RND, LRS and MRS5, RND5, LRS5.

In Figure 7.4, we investigate how the system performance is effected by different priority eviction policies. First, we investigate the effect of eviction policies on the system load, as a result number of occupied slots. In Figure 7.4(a), we show the

slot utilization which is defined as the percentage of occupied slots. When there are no available slots, i.e., 100% slot utilization, priority evictions start. The longer the duration of no available slot period, the larger the number of evictions and more wasted executions experienced. When the thresholding is applied since some of the tasks are dropped during peak arrivals, slots are freed earlier than MRS, RND and LRS. Since the evicted tasks reach the dropping threshold faster with LRS5 and RND5 than MRS5, LRS5 and RND5 shows similar slot utilization with MRS5 by exploiting task dropping. By selecting the youngest task to evict, MRS benefits from less additional workload due to reschedulings, which is also reflected as less number of tasks waiting in the queue as shown in Figure 7.4(b). However, LRS5, RND5 and MRS5 experiences queueing for shorter amount of time with the expense of dropping tasks. Queueing time is reflected to the number of on servers, as a result no available slots for a longer period of time. In Figure 7.4(c), we see that MRS, MRS5, RND5 and LRS5, schedule the queued tasks faster than RND and LRS and provide opportunity to make some servers to sleep earlier. Overall, MRS provides more slot availability hence less queueing by reducing wasted executions compared to RND and LRS. However, when dropping is allowed, RND and LRS take advantage of reaching threshold faster than MRS, and improve slot availability with the cost of dropping more tasks.

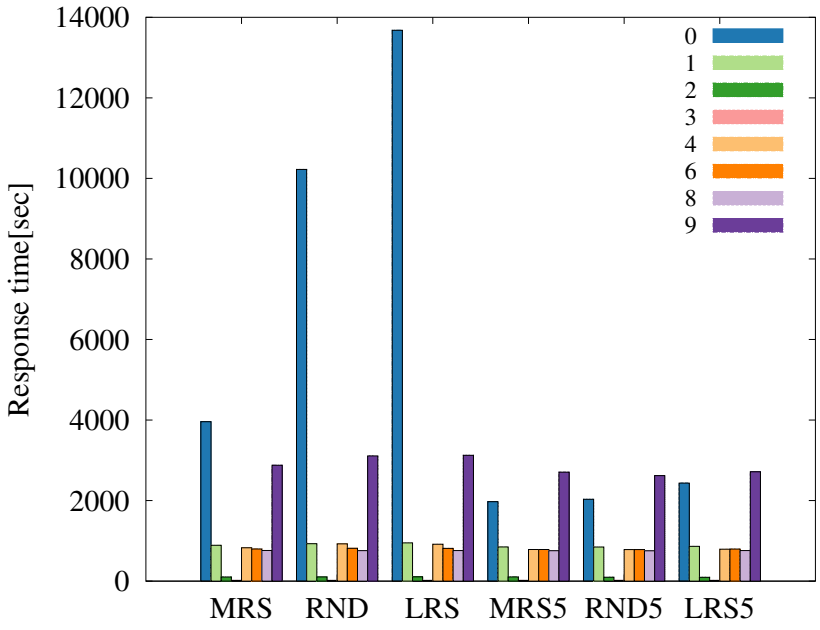


Figure 7.5. Response time by priority class with MRS, RND, LRS and MRS5, RND5, LRS5.

Lastly, we analyze the class-based response time in Figure 7.5. When compare LRS, RND and MRS, MRS provides the lowest class response time and LRS performs the highest. The difference on response time of the other classes by policy is not significant as class 0, but MRS improves the response time of almost all classes by minimizing the wasted executions. However, when thresholding is applied, lowest class response time is tremendously improved, by eliminating the straggler tasks, for LRS5, MRS5 and RND5. Overall, MRS5 gives the best class-based response time by providing low response time for high priority classes, reasonable response time increase in the lowest priority class and with 1.8% dropped tasks. From our results, we observe that applying thresholds can effectively improve slot availability hence reduce evictions significantly by eliminating straggler tasks from the system, which are more common with LRS and RND than MRS.

7.4. Chapter Summary

This study presents a comprehensive study of priority scheduling and evictions on large computing clusters, which aims to satisfy performance objectives of highly heterogeneous requests. We develop a slot-based priority scheduling technique as well as a number of eviction policies, i.e., MRS, RND and LRS and evaluate them with Google cluster trace. To mitigate the wasted resources and response time degradation, we propose to impose a threshold on the number of evictions and introduce task dropping. Our results show a significant improvement on the resource efficiency of priority scheduling, especially in terms of amount of wasted resources and response times of low priority tasks. Our key findings are that certain eviction policies lead to repetitive evictions of some tasks which cause outliers in response times.

8. WORKLOAD-AWARE PRIORITY SCHEDULING

Assigning tasks onto slots is one of central operations in slot-based resource assignment, and its performance depends on the slot configuration. Increasing number of slots enlarges the instantaneous scheduling capacity of the system, hence decreases queueing time. However, increasing the level of resource sharing or consolidation may result in resource scarcity hence slow down on execution times. On the other hand, different priority classes have different resource requirements. It is very challenging to satisfy the priority class-based performance levels by providing homogeneous slots.

The workload heterogeneity is a key factor to be considered in order to optimize task scheduling. Hence, we propose a task scheduling algorithm, which includes workload-awareness on both slot configuration and task assignment since workload oblivious approaches lead to non-optimal decisions due to incompatible resource requirements and allocations. Workload-Aware Slot Configuration and task Assignment (WA^{2}SC) consists of two parts: heterogeneous slot configuration and task assignment. The first step is done offline by using statistical properties of the workload, while workload-aware scheduling policy is integrated into the scheduler as explained in the following section. Our experimental results show that we are able to achieve up to 29% improvement on response times by choosing the right eviction policy, namely MRS (most recently started), and integrating workload-awareness to the slot configuration and task assignment.

In particular, we define powerfulness of a slot in terms of CPU since the response time of a task heavily depends on allocated CPU rate. Consequently, we define heterogeneity on slots based of average CPU capacity per slot, when fully loaded. Although the slots are inherently heterogeneous due to the resource heterogeneity of servers, by using CPU-based slot configuration, we provide CPU homogenous slots. Similarly, we refer powerful slots as the ones with higher average CPU capacity per slot, as described in Table 8.1. In the following sections, we compare proposed optimal workload-aware slot configuration and task assignment algorithm WA^{2}SC with workload oblivious pol-

icy employing different CPU-based homogenous slot configurations. WA^{2}SC provides powerful slots and assigns tasks with high resource demands to powerful slots. We first analyze how the number of evictions, resource utilization, in terms of CPU and memory and slot availability are changing with different slot configurations. Then, we focus on the improvement on class-based response time by the introduction of workload-awareness in slot configuration and task assignment with WA^{2}SC.

8.1. Workload-aware Slot Configuration and Task Assignment

WA^{2}SC relies on providing heterogeneous slots to meet the varying resource requirements of classes, and especially assigning tasks with high resource demands to powerful slots. Two key parts are workload-aware slot configuration (WASC) and workload-aware scheduling policy (WASP).

8.1.1. Workload-aware Slot Configuration

Here, we design a workload-aware slot configuration algorithm as shown in Algorithm 8.1, to determine types and number of slots providing a better fit to the heterogeneity of the workload. In order to make our system workload-aware, we use a statistical analysis of 7 days of the Google Cluster trace.

High priority, i.e., production class, tasks are rare but uses more resources as shown in Table 5.2. Intuitive solution is to assign production class tasks with high resource demands to fast cores [102]. Although few number of servers are holding fast cores in our server environment, assigning production tasks to fast cores does not improve their response time significantly since the resource sharing is determined by number of slots not by core capacity especially during peak load. In order to improve the response time of production tasks, we need to assign more resources to production class tasks in addition to the scheduling priority. Our approach is based on discovering heterogeneity level of the workload in terms of resource demand and configuring servers accordingly. Hence, we propose WASC algorithm to adjust the number of slots for production and non-production tasks. The algorithm is composed of two phases.

Phase I:

for each priority class i **do**

find CPU occupancy φ_i by multiplying P_i with ω_i, as Eq. 8.1

compute percentage of CPU occupancy ϕ_i, as Eq. 8.2

compute estimated CPU usage θ_i, as Eq. 8.3

end for

find percentage of CPU load for (non-)production classes ϕ_p and ϕ_{np}

estimate the CPU usage for (non-)production classes θ_p and θ_{np}

find slot configurations $[N_p N_{np}]$ for (non-)production classes, as Eq. 8.4

Phase II:

Sort servers according to the core capacity in decreasing order

for each server l **do**

$\beta \leftarrow$ percentage of CPU resource of l,

if $\beta > 0$ **then**

configure l with N_p^l number of slots

$\phi_p = \phi_p - \beta$

end if

configure l with N_{np}^l number of slots, as Eq. 8.5

end for

Figure 8.1. Workload-aware Slot Configuration (WASC).

8.1.1.1. Phase I. The objective of the first phase of the WASC algorithm is to find the distribution of CPU resources among production and non-production classes, and determine the CPU usage levels of each class. In order to find how much CPU resources are occupied by each priority class, we calculate the CPU occupancy of each class (φ_i), defined as the total CPU resources allocated to a priority class. While CPU usage (ξ_i) is defined as the average CPU rate used, CPU demand (ω_i) is the total CPU time required to execute. The CPU occupancy of a class is calculated by multiplying mean CPU demand and percentage of task arrivals (P_i) of the class as shown in Equation 8.1. Then, we calculate percentage of CPU occupancy (ϕ_i) of (non-)production classes,

according to the total CPU occupancy of (non-)production tasks, which designates how the CPU resources are shared among these two groups. According to the percentages of CPU occupation of (non-)production tasks based on Equation 8.2, we determine the percentages of CPU capacity to be allocated for (non-)production tasks.

$$\varphi_i = P_i * \omega_i \tag{8.1}$$

$$\phi_i = \frac{\varphi_i}{\sum_{j=0}^{K} \varphi_i} 100 \tag{8.2}$$

After finding the percentage of CPU to be used by (non-)production classes, we find the optimal number of slots suited for (non-)production classes by estimating the CPU usage for each priority class as shown in Equation 8.3. We use Chebyshev's inequality to guarantee the minimum coverage of the entries for unknown distributions [103], where θ_i is the estimated CPU usage of priority class i and σ_i is the standard deviation of CPU usage of priority class i. In this study, we choose confidence interval indicator c as 2 for (non-)production classes which covers for 75% of the entries at minimum. We calculate the estimate as shown in Equation 8.3. We also apply a slow down sensitivity factor δ according to the priority groups. Production classes with $\delta = 1.0$ are highly sensitive to slow down where gratis classes with $\delta = 0.5$ are not so sensitive and middle classes with $\delta = 0.75$.

$$\theta_i = \delta_i(\xi_i + c_i * \sigma_i) \tag{8.3}$$

At the last step of the first phase, we compute the number of slots for (non-)production tasks, according to the estimated CPU usage, using the server holding maximum CPU (type D in our case) as shown in Equation 8.4. Since all the resources are normalized according to the capacity of the most powerful server (C), the findings can be easily scaled to different server types with different CPU capacity (C_l) as shown

in Equation 8.5.

$$[N_p N_{np}] = C \odot [\frac{1}{\theta_p} \ \frac{1}{\theta_{np}}] \tag{8.4}$$

$$[N_p^l N_{np}^l] = \frac{C_l}{C} \odot [N_p N_{np}] \tag{8.5}$$

8.1.1.2. Phase II. In the second phase of the algorithm, we map the slot configurations of production (p) and non-production (np) classes (N_p, N_{np}) found in the first phase, to a heterogeneous server environment. In the first phase, we find that ϕ_p as the percentage of CPU occupancy of production classes, in this phase we configure ϕ_p of CPU resources with N_p slots. In order to provide simplicity, N_p and N_{np} are calculated as the multiples of minimum number of cores.

First, we sort the servers according to the speed of their cores. Starting from the server type holding fastest cores, we configure ϕ_p of the CPU resources with N_p^* slots for production classes. The rest of the servers are configured with N_{np}^* slots. Note that N_p^* and N_{np}^* denotes the scaled values of N_p^* and N_{np}^* for each server type according to Equation 8.5.

Table 8.1. CPU-based versus workload-aware system configurations.

		CPU-based					
		SC1	SC2	SC3	SC4	WASC	
Server	quantity	slots	slots	slots	slots	slots	CPU
A	68	8	16	24	32	24	normal
B	39	8	16	24	32	24	normal
C	10	8	16	24	32	24	normal
D	5	16	32	48	64	24	powerful
	3	16	32	48	64	48	normal
total	125	1064	2128	3192	4256	3072	-

When we apply WASC algorithm to our server environment, we obtain the following slot configurations which are shown in Table 8.1. The WASC algorithm results in two types of slots. The majority of the servers are configured with normal slots like in CPU-based slot configuration. Moreover, five servers of type D are configured with *powerful* slots in order to accommodate production classes which are really low in percentage but require more resources.

The proposed WASC algorithm requires statistical data which is gathered from historical workload characteristics. Mainly, we need to know the average CPU usage and demand for each priority class. We obtain the CPU demand of each task by multiplying its average resource usage and execution time. The workload statistics that are used to make system configurations can be collected online and system can be configured periodically or based on the changes in the characteristics of the workload.

8.1.2. Workload-aware Scheduling Policy

Upon slot configuration, task assignments are carried by "workload-aware scheduling policy (WASP)" as shown in Algorithm 8.2. Essentially, for the production classes, scheduler sorts the slots according to CPU and assigns the task to the most powerful slot to satisfy their high resource demands. If the tasks is not from production classes, it is assigned to a randomly chosen slot. The main idea is to match high resource demanding tasks with *powerful* slots. Consequently, if an eviction is required for a task from production class, the most powerful slot is freed which is occupied by the most recently started lowest priority. Thus, we ensure to assign most powerful slots to the high resource demanding tasks, to execute faster and avoid memory evictions. WASP algorithm provides privilege to production classes not only for scheduling order but also for resource assignment. Since the algorithm is based on ordering of tasks according to priority class, and slots according to powerfulness, it can be applied to any number of task classes and slot types.

```
for each task o in the queue do
  if there is an available slot then
    workload-aware assignment (WAA):
    if o is from production class then
      assign to the most powerful slot
    else
      assign to a randomly selected slot
    end if
  else
    workload-aware eviction (WAE):
    if o is from non-production class then
      apply MRS policy to the task occupying the most powerful slot
    else
      apply MRS policy
    end if
    add evicted task to the central queue
    assign o to the freed slot
  end if
end for
```

Figure 8.2. Workload-aware Scheduling Policy (WASP).

8.1.3. Complexity of WA^{2}SC

The WA^{2}SC is composed of two parts. The first part of WA^{2}SC algorithm WASC is done offline where WASP algorithm is online. The complexity of the proposed WASC algorithm depends on the size of the historical data which is $O(K)$ where K is the number of task statistics. The analysis of WASC depends on averaging and calculation of standard deviations. Hence its complexity is $O(K)$. The complexity of the proposed online WASP algorithm depends on the sorting algorithm applied for sorting the tasks according to their priority. Hence, complexity of the algorithm is $O(NlogN)$ as sorting

can be done by an algorithm with the order of $O(NlogN)$ complexity. Overall, our proposed algorithms are computationally efficient as they can be solved in polynomial time.

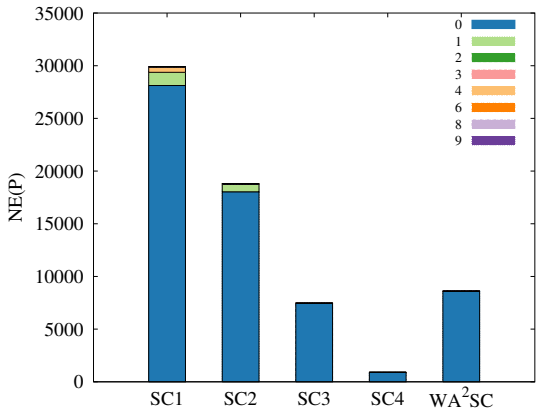
8.2. Performance Evaluation

We compare WA^{2}SC and workload oblivious policy employing different CPU-homogeneous slot configurations. Workload oblivious policy works based on slot-based priority scheduling and employs MRS and LSF eviction policies which analyzed in Section 7.3.4, and find out to be the best in terms of class-based response time without dropping tasks. In Table 8.1, we show slot configuration based on CPU capacity (SC) which is the default configuration and the slot configuration found with Algorithm 8.1. We refer experiments with workload oblivious policy by the slot configuration, e.g., SC1 refers to the experiment with workload oblivious policy with SC1 slot configuration. In particular, we analyze our proposal with Google cluster environment with 15 hour long simulation, which we also use in the previous section.

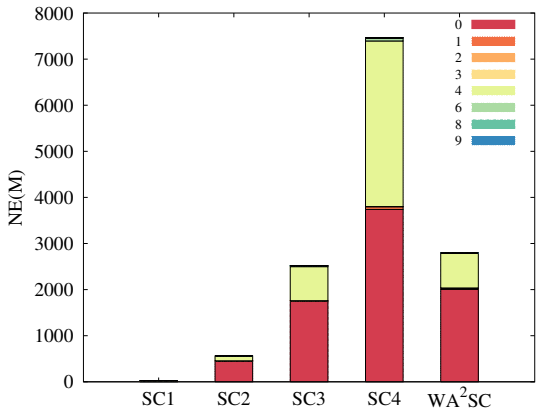
In our analysis we investigate the improvement on performance, i.e., class-based response time and resource utilization with different slot configurations with a special attention paid to memory and priority evictions.

8.2.1. On Evictions: Priority Evictions vs. Memory Evictions

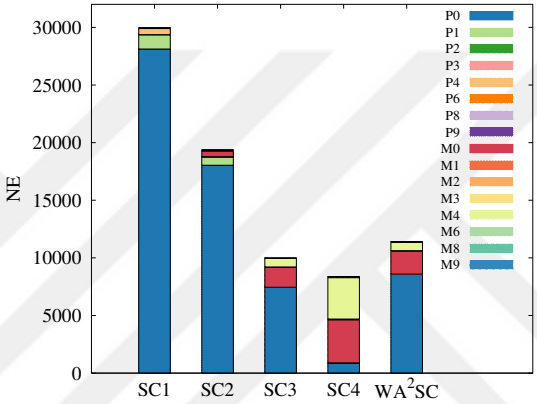
We first analyze how the number of priority and memory evictions changes with different number of slots in Figure 8.3. As expected number of priority evictions rapidly decreases as the number of slots increases as can be seen in Figure 8.3(a). With SC1, priority evictions are so high and evictions are observed by not only class 0 but also, 1, 4 and 6. However, when the number of slots is increased, i.e,. SC3 and SC4, priority evictions are only experienced by class 0. We can see that priority evictions are highly dependent on the number of slots in the system. WA^{2}SC follows a similar trend with SC3 since the total number of slots in the system is almost same (3.75% less). When the number of slots is configured as SC4, the system can operate with insignificant number



(a) Number of priority evictions.



(b) Number of memory evictions.



(c) Total number of evictions.

Figure 8.3. Number of priority and memory evictions for WA^{2}SC and workload oblivious policies SC1, SC2, SC3, and SC4.

of priority evictions at a high cost of memory eviction due to extensive resource sharing as shown in Figure 8.3(b). With the increasing number of slots, memory capacities are often exceeded resulting in memory evictions. As the number of slots become 32 with SC4, we observe memory evictions in class 6, 8 and 9, while no significant number of evictions with SC1. Although the number of memory evictions with SC3 and WA^{2}SC are similar, WA^{2}SC results in less number of evictions in priority 6 and 8, due to its assignment of classes with high resource demands to powerful slots. It is worth noting that, high priority tasks are not detrimentally effected by memory evictions like low priorities since they are scheduled in the next slot with the first priority, even causing priority evictions.

When we sum up priority and memory evictions, priority evictions are more

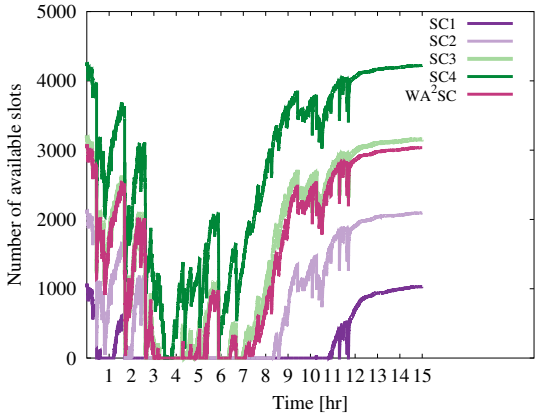
dominant compared to memory evictions as shown in Figure 8.3(c). Moreover, as the number of slots increases the part that memory evictions takes also increases and exceeds the number of priority evictions by SC4. Although increasing number of slots helps to decrease number of evictions, the improvement is not significant between SC3 and SC4 slots. WA^{2}SC experiences similar number of eviction with SC3, and less number of high priority memory evictions, by the assignment of high resource demanding classes to powerful slots. Overall, WA^{2}SC outperforms homogeneous slot configurations by balancing the tradeoff between priority and memory evictions, via adapting slot configurations according to workload characteristics.

8.2.2. On System Utilization

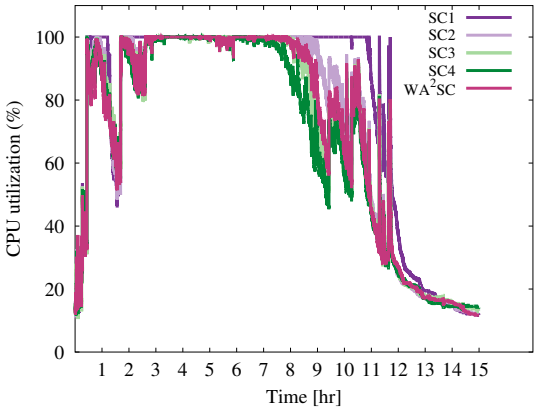
One of the main concerns when configuring slots is the system and resource utilization. We can keep the number of slots low and avoid memory evictions, however this may lead to poor resource utilization and increase in priority evictions. In some cases, although the system looks full with low number of slots, CPU/memory resources might be underutilized. In these analysis our key performance metrics are CPU and memory utilization, which is the CPU/memory usage over total capacity over time, in addition to the slot availability. Additionally, homogenous slot configurations may result in low resource utilizations with the workloads with diverse resource requirements. Thus, we try to find a good operating region to avoid underutilization and provide heterogeneous slots to better fit tasks to slots.

We analyze slot and resource utilization for changing number of slots in Figure 8.4. The first observation from Figure 8.4 is, even though the total number of slots differs, number available of slots shows similar pattern as shown in Figure 8.4(a). Although the change in number of available slots follows the task arrival pattern, lower number of slots suffers from lack of available slots and long queues. For SC1, the system stays longer with no available slots, however for SC4 the number of available slots becomes 0 only once. The pattern of number of available slots with WA^{2}SC stays almost same with SC3 since the total number of slots are similar. Hence, we can conclude that with low number of slots the system looks fully occupied in terms of slots, which triggers

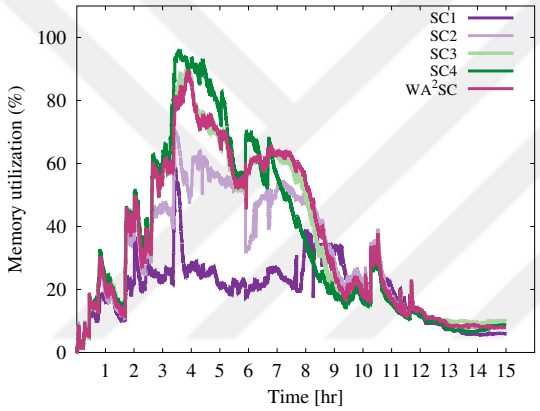
priority evictions, although memory is underutilized.



(a) Number of available slots.



(b) CPU utilization.



(c) Memory utilization.



(d) Normalized energy consumption.

Figure 8.4. Number of available slots and CPU, memory utilizations, energy consumption for WA^{2}SC and workload oblivious policies SC1, SC2, SC3, and SC4.

In addition to the slot occupancy, we also investigate the utilization of CPU and memory in Figure 8.4(b) and Figure 8.4(c). Figure 8.4(b) confirms that WA^{2}SC allows running tasks to exploit extra CPU/core capacity of the server. Therefore, we achieve almost full utilization when the load is high, however high number of slots benefits from finishing the bursts faster by avoiding priority evictions that causes an increase in the load. Another key observation is that, memory utilization with low number of slots is lower than high number of slots, although their slot and CPU utilizations are high. The reason behind low memory utilization with low number of slots is that the memory usage is not flexible like CPU, hence low number of slots leads to memory underutilization. Although we achieve significant improvements on response time and

number of evictions evictions with WA^{2}SC, energy consumptions of different configurations are not significantly different. The low memory utilization of SC1 is reflected as a 10% less energy consumption compared to SC3, SC4 and WA^{2}SC which perform very similar as shown in Figure 8.4(d). The lower energy consumption of SC1 comes with a cost of significantly high response time of low priorities and higher wasted executions.

In summary, low number of slots, i.e., SC1 and SC2, suffers from low memory utilization and scarcity of available slots especially during the peak load. On the contrary, by providing sufficient number of heterogeneous slots WA^{2}SC provides better slot availability and CPU/memory utilization at the same time compared to workload oblivious schedulers.

8.2.3. On Response Time

The primary performance metric, per class response time, highly depends on the allocated CPU, experienced evictions and queueing time. Since number of slots is directly or inversely proportional to these factors, we show how per class response time changes with number of slots and different slot configurations in Figure 8.5. The average response time improves with the increasing number of slots, in spite of the increase on high priority class response time. Increasing number of slots improves lowest priority class response time significantly by eliminating priority evictions and long waiting times. On the other hand, high priority classes suffer from decreased CPU allocation due to increased number of co-executing tasks. Hence taking R as the primary performance metric may lead to misleading decisions.

8.2.3.1. Impact of Class Arrivals on R. R is not a perfect metric to quantify the performance in multi-class systems with varying number of arrivals since it can not capture the effect of priorities and it is highly dependent on the class population. Hence, we propose a more sophisticated metric V which is the averaged weighted response time, where the weights are the priority classes and independent of the number of arrivals of classes as shown in Equation 6.5. When we compare R and V, up to SC4, V is

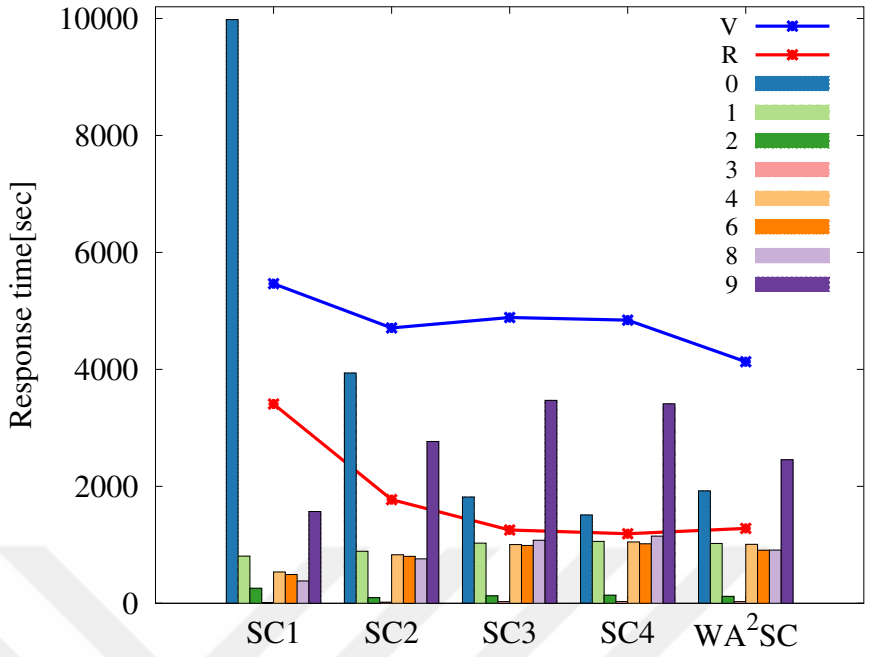
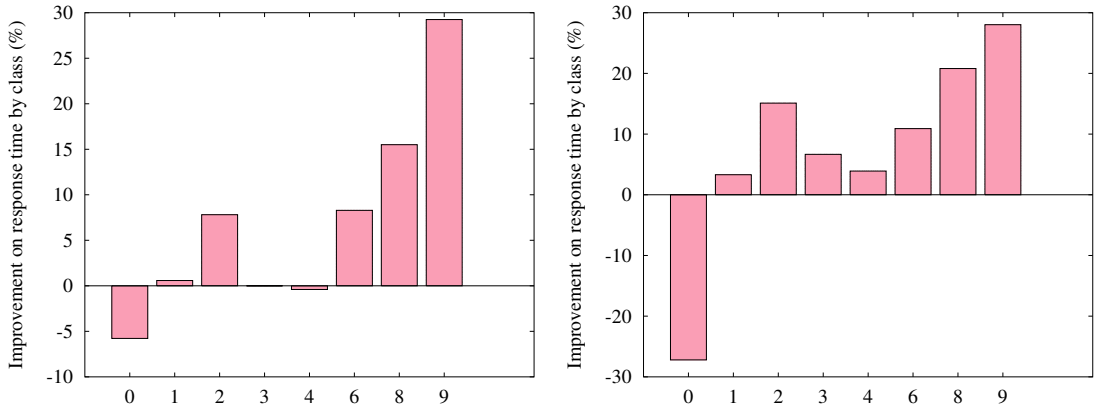


Figure 8.5. Response time by priority class, V, R.

increasing while R is decreasing. Improvement on R is a result of the improvement on lowest class since V is population independent. However, degradation on high priorities dominates the improvement on low priority response time. Overall, WA^{2}SC improves V by 18% over SC3.

8.2.3.2. Improvement on Class-based Response Time. When we compare SC1 and SC2 with SC3 and SC4, we see that low priority response time is getting better with the increasing number of slots with the expense of an increase in high priority response time as shown in Figure 8.5. Moreover, we can not obtain further significant improvement on low priority or high priority response time by increasing number of slots which is also visible from similar V values. However, we can improve both high and low priority response time with WA^{2}SC compared to SC3, by introducing workload-aware slot configuration and assignment with a reasonable slow down in the lowest priority class.

In Figure 8.6, we look closer to the improvement on average response time of each class, compared to SC3 and SC4. While comparing our proposed methodology WA^{2}SC with SC3, we obtain improvement almost every class. The amount of improvement lies between -6% and 29%. It is worth noting that, the improvement on high priority



(a) Improvement on response time by class: WA^{2}SC versus SC3.

(b) Improvement on response time by class: WA^{2}SC versus SC4.

Figure 8.6. Improvement on class-based response time with WA^{2}SC compared to workload-oblivious policies SC3 and SC4.

class response time is significant, i.e., 8%, 16% and 29% improvement for class 6, 8 and 9 respectively without a significant degradation on low priority response time. Since the aim of WA^{2}SC is to provide powerful slots to high resource demanding classes, the improvement on response time of high priority classes supports the objective. We also compare WA^{2}SC with SC4 to analyze if we can gain more by increasing number of slots. The first observation from Figure 8.6(b) is that WA^{2}SC still provides notable improvement (28%) on the response time of almost all classes with the cost of 27% increase in the response of class 0. While the improvement is obtained by providing powerful slots to high resource demanding tasks, the degradation on the response time of class 0 is caused by the increased number of priority evictions of class 0 as explained in Figure 8.3(a).

WA^{2}SC algorithm is designed to meet the highly heterogeneous demands of priority classes. While WA^{2}SC utilizes the advantages of variability for highly heterogeneous workloads, it is applicable to workloads holding less variability on resource demands by determining the slots according to workload characteristics. The scale of the improvement with WA^{2}SC changes with the load. The highest improvements for V are obtained from the low workload intensity cases. For high load when the system is highly utilized, the opportunity to improve the performance is smaller. The algorithm works with homogenous (identical) servers as well.

The efficiency of the WA^{2}SC algorithm lies at proactively configuring slots according to the workload characteristics and assigning tasks to slots based on resource demand. Overall, WA^{2}SC outperforms workload oblivious policies and improves high priority class response time by integrating workload-awareness in both slot configuration and task assignment.

8.3. Chapter Summary

Motivated by the highly varying demands of tasks for resources, we propose to optimize task scheduling by introducing workload-awareness in slot configuration and task assignment. Our results show a significant improvement on the resource efficiency of priority scheduling especially in terms of response times of high priority tasks compared to workload oblivious approaches. The proposed algorithm aims to provide an adequate number of heterogeneous slots such that high priority tasks are executed on powerful slots and burst arrivals can be served immediately. Our results indicate that workload-awareness offers a lot of potential for highly heterogeneous workloads having diverse and unknown resource demands.

9. HYBRID SCHEDULING POLICY FOR PRIORITY-BASED OPERATION OF COMPUTING CLUSTERS

In Chapter 8, we propose a workload-aware priority scheduling system. The proposed system provides significant improvements on resource efficiency and class-based response time. However, in preemptive priority scheduling systems low priority tasks may face with long queueing times and resource starvation due to long running high priority tasks. Hence, in this chapter we investigate the inefficiencies of preemptive priority scheduling in terms of long latency of low priority tasks and consequently high resource waste from repetitive execution of evicted tasks. Essentially, we focus on providing scheduling and execution guarantees for low priorities while preserving performance levels of high priorities. Hence, we propose a new scheduling policy called HYBRID which can resolve such a pitfall while preserving the performance benefit of high priority task. In particular, we improve not only the response time of low priority tasks but also the resource efficiency of the cluster in terms of reduced wasted resources.

9.1. Motivation

Preemptive priority scheduling has become widely used in large clusters executing heterogeneous workloads to fulfill the requirements of different workloads [2, 5, 7]. However, preemptive priority scheduling may lead to serious performance inefficiencies, i.e., schedulability, resource starvation problems and outlier response times especially for low priority tasks. The problem of cluster resources "hogged" with a large number of long-running tasks from high priority classes result in long scheduling delays, lack of resources for low priorities due to long occupation of high resource demanding high priorities, repetitive evictions. Approaches like dynamic prioritization [104] can solve schedulability problem which is necessary, but not sufficient for starvation-free system when evictions take place. Providing dedicated resources for each class may guarantee execution of every class however, it is very inefficient in terms of resource

utilization [105] and it is hard to estimate resource usages of each class beforehand [3].

To achieve priority scheduling with no starvation problem for prioritized heterogeneous workloads, we propose to control task assignments in combination with preemptive priority and non-preemptive first come first serve (FCFS) scheduling. We leverage the idea of providing service to long waited low priority tasks with FCFS scheduling policy with the guarantee of non-preemptive execution while preserving the performance levels of high priorities with preemptive priority scheduling. To such an end, we propose HYBRID scheduling policy which schedules tasks in combination of preemptive priority and non-preemptive FCFS. In addition to improvements on response time of low priorities, we aim to reduce wasted resources hence improve resource efficiency by mitigating repetitive evictions. Essentially, the task response time of a task depends not only on queueing time which is related to schedulability, but also on evictions experienced which is related to probability of successful execution.

In particular, HYBRID consists of two phases: the first phase schedules tasks in FCFS order with a non-preemptive scheduling discipline and the second phase employs preemptive priority scheduling for remaining tasks in the central scheduling queue. HYBRID introduces a new type of slot called *sticky slot* which provides uninterruptible execution of tasks. Sticky slots are only available for the first phase of the scheduling. In addition to the improved scheduling opportunity for low priority tasks, HYBRID improves the rate of successful execution of low priorities by utilizing sticky slots. Using trace driven simulations, we evaluate HYBRID on Google cluster trace and with synthetic and semi-synthetic traces. Our results show that HYBRID not only provides better response time for low priority tasks especially evicted tasks but also leads to a noteworthy decrease on wasted resources compared to a scheduling policy that does not provide starvation-free system. In a nutshell, HYBRID achieves efficient execution of heterogeneous workloads by successfully eliminating outlier response time hence providing more uniform performance for low priorities. HYBRID harvests the resource efficiency due to reduced wasted executions by the significant drop on the number of priority evictions.

Our contributions can be summarized as follows:

- we employ two phase scheduling policy to solve schedulability and repetitive eviction problem of long waited low priority tasks,
- the proposed HYBRID controls the degree of non-preemptiveness of the system by limiting the number of sticky slots,
- HYBRID limits the competition of resources between high and low priorities by utilizing limited number of sticky slots which provides a flexible and virtual resource isolation,
- our evaluation on traces obtained from production systems show promising gains on resource efficiency as well as improvements on response time of low priority tasks without degrading performance of high priorities,
- HYBRID provides more uniform response time for low priority tasks by mitigating repetitive evictions as well as less number of priority evictions.

9.2. HYBRID Scheduling Policy

In this chapter, we propose a novel scheduling policy called HYBRID which effectively handles task resource assignment for highly heterogeneous workloads to provide guaranteed execution of low priorities as well as giving precedence to high priorities. We build HYBRID scheduling policy on top of our proposed workload-aware system. In order to provide guaranteed scheduling and execution of low priorities HYBRID adopts two phases of scheduling and limits priority evictions by the introduction of sticky slots. Sticky slots work as regular slots except the tasks executing with sticky slots can not be evicted due to priority, i.e. sticky slots provides an uninterruptibility shield for priority evictions. In our system, fixed number of sticky slots reside in the system throughout the execution of the workload.

The scheduler needs to successfully make the following decisions: i) scheduling order: which task to schedule, ii) execution is interruptible or not: which slot to use as sticky or regular, iii) if there is no available slot in the system, whom to evict. In the first phase, the scheduler handles most disadvantaged tasks, mainly long waited and/or

evicted low priorities, in FCFS order by assigning them based on the resource (sticky slot) availability. These tasks are scheduled as uninterruptible in order to guarantee the completion of their execution. In the second phase, the scheduler works as a fully preemptive priority scheduler. The unscheduled tasks after the first phase of the scheduling are served according to priority, i.e., highest priority first. At this phase, the scheduler can evict a low priority task if the system is full in order to assign resources to a high priority task. MRS eviction policy is employed for priority evictions. The complexity of the proposed HYBRID scheduling is $O(NlogN)$ since the computational complexity is dominated by sorting of tasks in both phases. The outline of the system with HYBRID scheduling policy is shown in Figure 9.1.

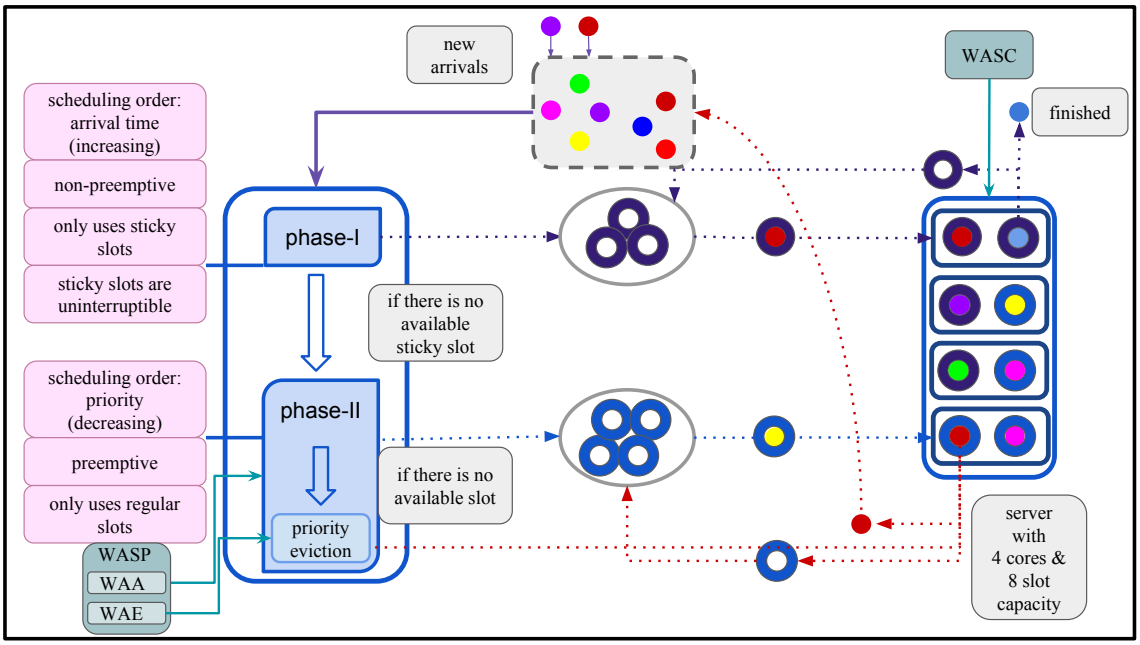


Figure 9.1. The system design with HYBRID scheduling. The system is configured according to workload-aware slot configuration (WASC) as shown in Algorithm 8.1. Phase II employs workload-aware Scheduling Policy (WASP) which is presented in Algorithm 8.2.

9.2.1. Phase-I

At each time slot HYBRID scheduler checks the central scheduling queue. If there are waiting tasks, the tasks are sorted in an ascending order by their arrival time to the system. There are two reasons behind this operation. First, the low priority tasks

can not get scheduled due to low scheduling precedence over high priorities. Second, repetitively evicted tasks are the most disadvantaged tasks, they incur queueing time, wasted executions at least once and evicted task have the earliest arrival time to the system among the same class tasks. We note that, the arrival time of an evicted task refers to the time stamp when the task first arrives at the system.

At this phase, the scheduler works as a first come first serve, non-preemptive scheduler. Phase-I utilizes sticky slots which are virtual labels and are not reserved on specific machines. The task scheduled with regular slots and sticky slots can run on the same machine. The sticky slot pool is only available for phase-I. Therefore, there is no competition with high priority tasks and no risk of waiting to be scheduled indefinitely due to priority and getting evicted due to priority for low priority tasks. Hence, our proposed system is safe in terms of schedulability. Essentially, phase-I not only guarantees schedulability of the low priorities but also guarantees uninterrupted execution after being scheduled.

Phase-I is non-preemptive so no task is evicted inorder to schedule another task. Additionally, the tasks which are scheduled at this phase are promoted to be uninterruptible. Uninterruptible tasks can not be evicted due to priority. When there is no waiting task or there is no sticky slot available HYBRID moves to phase-II. In particular, we use 200 sticky slots (6.5% of all slots) throughout the evaluations in this paper. We set the number of sticky slots based on our tradeoff analysis which effectively mitigates long waitings and repetitive evictions while not hurting high priorities. The detailed analysis of different number of sticky slots are presented in Section 9.3.3.

9.2.2. Phase-II

At phase-II, the scheduler utilizes WASP algorithm which is proposed in Section 8.1.2. The central scheduling queue is sorted according to the priority class in descending order and then according to the arrival time in ascending order. The scheduler dispatches the task from the head of the queue and schedules the task based on the regular slot availability. For the production classes, scheduler essentially sorts the

slots according to average CPU per slot and assigns the task to the most powerful slot to satisfy their high resource demands. If the tasks are not from production classes, it is assigned to a randomly chosen slot. The main idea is to match high resource demanding tasks with *powerful* slots. Consequently, if an eviction is required for a task from production class, the most powerful slot is freed which is occupied by the most recently started lowest priority. Thus we ensure to assign most powerful slots to the high resource demanding tasks to execute faster and avoid memory evictions. The priority evictions are handled same as WAE policy. At this phase the scheduler can not use sticky slots. Hence, there is no slot competition between phase-I and phase-II of the scheduling.

9.3. Performance Evaluation

We evaluate the HYBRID scheduling policy in comparison to the WASP policy which is explained in Section 8.1.2. We first seek to show the effectiveness of HYBRID scheduling on system performance mainly in terms of class-based response time and repetitive evictions, compared to WASP. Regarding the difference between HYBRID and WASP scheduling policies, WASP scheduling does not have a phase-I. Hence all the slots in the system are utilized by the priority scheduler and all slots in the system are eligible to evict. We obtain WASP scheduling using HYBRID scheduling policy if the number of sticky slots is set to zero thereby bypassing the phase-I.

For our simulation driven performance analysis, we used the workload and server environment configured with WASC (see Table 8.1) that we use in the previous chapter. Our system adopts slot-based resource assignment and slot configuration is done according to WASC algorithm. In particular, we analyze our proposal with Google cluster environment with 15 hour long simulation, which we also use in the previous chapters. We evaluate the performance of the HYBRID in terms of the class-based response time, response time of evicted tasks, number of priority evictions, number of evicted tasks, maximum number of priority evictions, wasted executions.

Table 9.1. Evictions, response time and WE analysis of HYBRID and WASP.

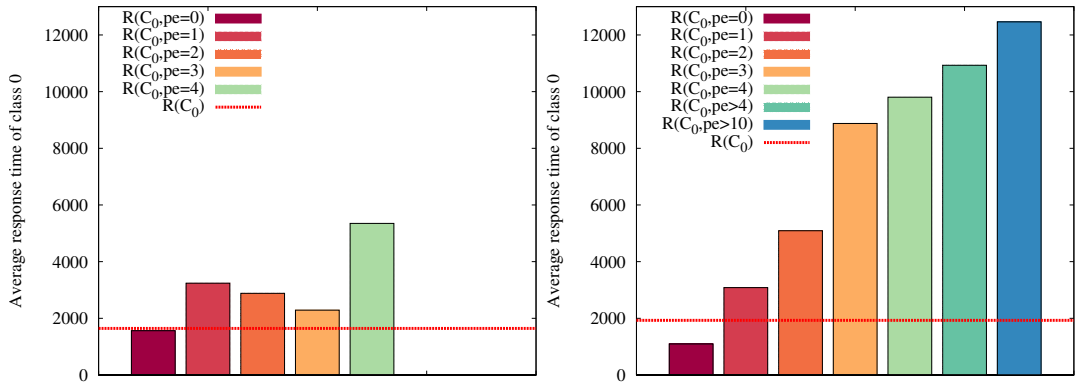
metric	HYBRID	WASP
NE	4540	11305
$NE_{priority}$	1231	8509
$NET_{priority}$	1044	3799
$\max(pe)$	4	14
$\overline{pe}$	1.1	2.23
$R(C_0)_{non-evicted}$	1562	1096
$R(C_0)_{evicted}$	3179	5601
$R(C_9)$	2409	2685
WE(10^9)[cpu.sec]	20.9	141
E[kWh]	404.4	416.9

9.3.1. Impact on Evictions

In priority based preemptive scheduling systems, the response time of low priority classes are severely affected by scheduling order and the eviction policy employed. The priority evictions do not only hurt the execution of the task which is result in waste of executions but also increase the system load by resubmissions of the evicted tasks. In Table 9.1, we look at the priority evictions and their effect on wasted executions and the response time of the lowest priority class where all of the priority evictions are experienced by class 0. The total number of evictions NE is decreased by 60% by HYBRID scheduling compared to WASP. Although the number of memory evictions increases slightly, number of priority evictions $NE_{priority}$ is significantly blocked by the introduction of sticky slots.

In addition to the significant decrease on $NE_{priority}$, HYBRID scheduler also improves average number of evictions per evicted task $\overline{pe}$ by 50% and significantly reduces the maximum number of priority evictions experienced $\max(pe)$ (repetitive evictions) by distributing evictions more uniformly. While the WASP scheduling policy imposes

absolute prioritization with the cost of increased system load and outlier response time of low priorities, HYBRID scheduling policy effectively handles both prioritization and performance issues of low priorities. Although we experience an increase on the average response time of non-evicted tasks from class 0 ($R(C_0)_{non-evicted}$), HYBRID scheduler successfully eliminates outlier response times of evicted tasks $R(C_0)_{evicted}$. $R(C_0)_{evicted}$ is almost halved by the use of phase-I scheduling. While HYBRID scheduler ensures good QoS for low priorities, the high priority tasks are even affected positively in terms of response time. This is because HYBRID scheduler reduces wasted resources WE tremendously (85%) which result in reduced system load and more resource availability. As a result, average response time of class 9 is improved by 10%. Although we achieve crucial improvements on response time and wasted executions, energy consumptions of two approaches are not significantly different. The reason behind is that reduction on wasted executions hence on system load with HYBRID is comparably smaller than the overall system load. Hence, there is very little room left to improve in terms of energy consumption. It is worth noting that, we greedily reduce the energy consumption of the system. We achieve significant energy savings via adopting sleep mode and MRS eviction policy.



(a) Average response time with HYBRID (b) Average response time with WASP

Figure 9.2. Impact of repetitive evictions on the response time of evicted tasks of priority class 0: HYBRID vs. WASP.

9.3.2. Impact on Class-based Response Time

In Figure 9.2, we investigate how the lowest priority class response time is affected by the number of repetitive evictions where $R(C_0, pe = x)$ denotes the average response

time of tasks that experience x number of priority evictions and $R(C_0, pe > x)$ denotes the average response time of tasks that experience more than x priority evictions. With WASP (see Figure 9.2(b)), the average response time consistently increases with the increasing number of evictions experienced. With HYBRID (see Figure 9.2(a)), we observe that the outlier response times are significantly reduced by the introduction of two-phase scheduling. Although WASP provides only 20% higher average response time for class 0 compared to HYBRID, the tail response time is significantly higher with WASP. While the max(pe) for WASP is 14 and max(pe) for HYBRID is 4 as shown in Table 9.1. HYBRID not only minimizes the max(pe) hence tail response time for the lowest priority class but also provides a much more uniform response time hence minimizes the performance disparity for tasks among the same class.

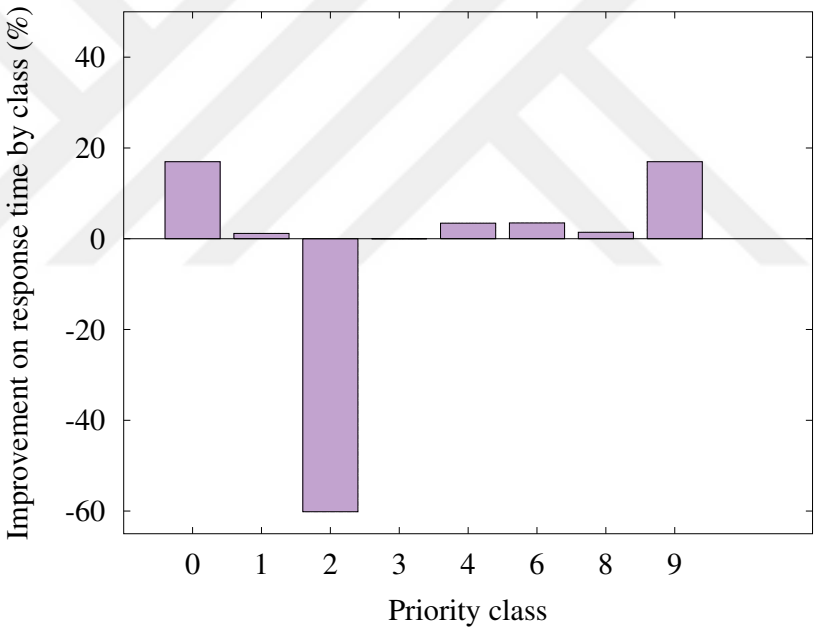


Figure 9.3. Improvement on class-based response time with HYBRID scheduling policy compared to WASP.

In Figure 9.3, we focus on the change of overall class-based response time by HYBRID scheduling compared to WASP. We observe that response time of almost all classes are improved. Phase-I gives service to long waited tasks mainly the low priorities and also guarantees uninterruptible execution of the scheduled tasks. As a result of the schedulability guarantee and elimination of resource starvation by the use of sticky slots, the response time of class 0 and 1 is improved. Especially, the response time of the evicted tasks are significantly improved and outlier response times are avoided as

shown in Table 9.1. HYBRID also works pretty good for high priorities since absolute prioritization is still possible at phase-II. The only negatively effected class is class 2. That is a reasonable trade off since the population of class 2 is really low (0.0004%). We can conclude that HYBRID scheduler effectively provides favorable response time for almost all classes.

9.3.3. Impact of the Number of Sticky Slots

The number of sticky slots in the system determines the degree of the preemptive priority scheduling and FCFS scheduling. When the number of sticky slots is set to 0 then the system works with WASP policy by disregarding phase-I scheduling. On the other hand, if we set number of sticky slots to total number of slots then the system works as a non-preemptive FCFS system while not using phase-II of the scheduling.

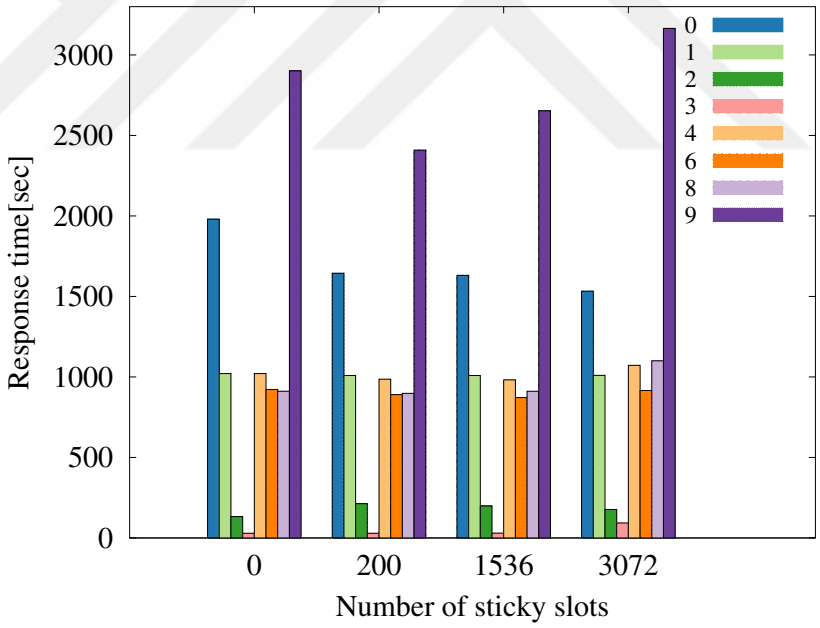


Figure 9.4. Class-based response time of HYBRID scheduling with 0, 200, 1536 and 3072 sticky slots.

When we compare HYBRID scheduler with 0 (0%), 200 (6.5%), 1536 (50%) and 3072 (100%) sticky slots, we observe the tradeoff between the high priority and low priority classes. Low priority tasks suffer from repetitive priority evictions when their execution is not guaranteed. If we increase the number of sticky slots from 0 to 200 we observe 20% improvement on average response time of class 0. However, increasing

the number of sticky slots further does not necessarily improve lowest priority response time even causing a significant increase on response time of production tasks. Overall, excessive number of sticky slots degrades the production tasks service quality on the other hand lack of sticky slots results in poor performance of low priorities.

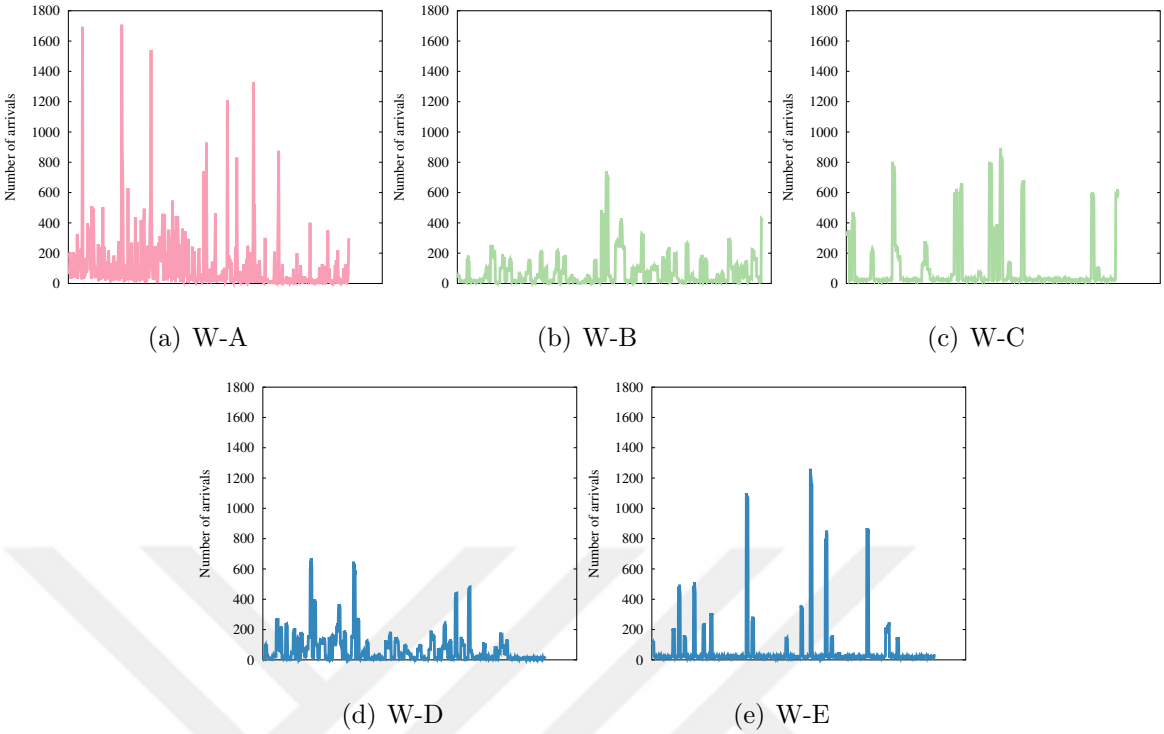
Table 9.2. Trace characteristics: real, semi-synthetic and synthetic traces.

Parameter	W-A	W-B	W-C	W-D	W-E
workload source	google	semi-synthetic	semi-synthetic	synthetic	synthetic
burstiness	very high	low	high	low	high
maximum number of arrivals in 1 min	1708	742	895	671	1261
maximum number of arrivals in 10 min	3802	3414	5113	3876	7285
maximum duration between two arrivals [sec]	122	261	32	360	51
burst frequency per hour	-	6.5	1.5	5.8	1.5
burst longevity	short	long	long	long	long

9.3.4. Evaluation of HYBRID Scheduling with Diverse Workloads

In order to investigate the effectiveness of our proposed system with HYBRID scheduling policy, we evaluate its performance with various workloads. We are mainly interested in the tradeoffs on class-based response time while providing execution guarantees. Therefore, we compare HYBRID scheduling with WASP scheduling with different workloads. We present the basic properties of the workloads in Table 9.2. We evaluate our system with synthetic (W-D, W-E) and semi-synthetic traces (W-B, W-C) in addition to the Google cluster trace (W-A). We also present the detailed arrival patterns of the workloads in Figure 9.5.

Both semi-synthetic and synthetic traces utilizes Poisson arrival process. While generating synthetic arrivals, we adjust the task arrival rate based on the desired total



(a) W-A (b) W-B (c) W-C

(d) W-D (e) W-E

Figure 9.5. Arrival patterns of 5 different trace with 60K task arrivals in 15 hours.

number of tasks. We show the burstiness of the workloads parameterized by the peak of the burst, duration of the burst and frequency of the burst arrivals. In order to achieve various burstiness levels, we control the mean of exponentially distributed inter-arrival times while synthetically generating random traces. The burst interarrivals, burst size and burst longevity are exponentially distributed. The mean values are adjusted according to desired burstiness level.

For semi-synthetic traces, we sample resource usage values from the real trace while generating the task arrivals. Hence, we offer to preserve highly diverse class resource usage properties for varying task arrival patterns. Moreover, for synthetic traces W-D and W-E, CPU and memory usages are exponentially distributed where productions have significantly higher mean. Task execution length is also exponentially distributed. The population of each priority class is determined by the probability distribution of classes in the real trace. The parameters used to generate semi-synthetic and synthetic traces are shown in Table 9.4.

In Table 9.3, we evaluate HYBRID scheduling compared to WASP with a real

Table 9.3. Improvement on class-based response time with HYBRID [%] compared to WASP.

	Priority class							
workload	0	1	2	3	4	6	8	9
W-A	17.29	0.10	-56.85	0	3.72	3.36	4.66	26.85
W-B	1.08	-0.44	-24.49	-41.38	-1.16	3.94	-4.13	16.09
W-C	14.29	2.31	-6.03	25.00	-1.90	1.05	4.66	12.54
W-D	0	-0.67	2.6	0	0	10.83	0.12	-3.24
W-E	10.23	-3.50	-2.69	-8.91	-2.43	0.46	-0.59	2.15
W-A_l	0.73	0	19.83	0	0	-1.84	0.96	21.77
W-B_l	0.94	0.70	-0.00	18.18	0.31	-2.28	-4.78	-2.33
W-C_l	0.34	-0.17	3.77	0	-0.14	3.29	2.31	6.87
W-D_l	1.10	-0.73	-17.75	0	0.73	0.73	0.30	1.80
W-E_l	7.14	0	16.82	-35.92	-2.72	0.25	-1.10	-0.66

trace W-A, two semi-synthetic traces W-B and W-C and two synthetic traces W-D and W-E. We show the improvement on class-based response time with HYBRID scheduling policy. In such a system, we expect a strong connection between burstiness and response time. As the burstiness increases, it gets harder to satisfy the performance goals of all priority classes as well as providing guaranteed execution for all priorities. However, our proposed HYBRID approach successfully deals with both. Almost for all traces, HYBRID approach improves response time of class 0 with a possible expense of an increase on response time of middle priority classes. Although the increase on response time of middle priority classes might seem significant for some workloads, the population of the negatively effected tasks are really low (<0.001%).

We also evaluate our system with lower loads which are approximately 25% lower load than the workloads shown in Table 9.2. We denote low load version of each trace with W-$*_l$. Although low load workloads follow same arrival patterns and resource usages with high load workloads, we do not observe significant improvement on low

priority response time. The reason behind is that these workloads do not experience excessive number of priority evictions which degrades the system performance. Hence there is very little room to improve for HYBRID scheduling.

Overall, HYBRID provides better low priority response time with a reasonable increase on middle priority response times for all workloads. While WASP approach is designed for giving utmost importance to prioritization other than providing guaranteed execution for all classes, it fails to adapt its scheduling policy based on system dynamics. On the other hand, HYBRID approach aims to provide opportunity and resources for long waited tasks mainly low priorities and prioritization for high priorities in the meantime. HYBRID successfully achieves that goal by limiting the competition of resources between high and low priorities by utilizing sticky slots for flexible and virtual resource isolation.

9.4. Chapter Summary

Motivated by the performance disparities between evicted and non-evicted tasks of the same class, we propose HYBRID scheduling policy. HYBRID aims to provide scheduling and execution guarantees for low priorities while preserving the performance benefits of high priority tasks simultaneously. Overall, HYBRID orchestrates preemptive priority scheduling and non-preemptive FCFS scheduling to improve scheduling opportunity of long waited low priority task as well as their successful execution. Using on-production trace and synthetic traces, our simulation results on a heterogeneous cluster show that HYBRID is able to achieve improvement on the response time of low priorities without causing degradation on high priority performance compared WASP policy which does not provide guaranteed execution and scheduling. The success of HYBRID lies at providing limited number of sticky slots for non-preemptive FCFS scheduling, thus leading to a significant reduction on repetitive evictions as well as the wasted resources. Overall, HYBRID improves cluster resource efficiency in terms of wasted resources and response time of low priority tasks by mitigating the number of priority as well as repetitive evictions.

Table 9.4. Parameters used to generate semi-synthetic and synthetic traces.

Parameter	W-A	W-B	W-C	W-D	W-E
mean interarrival time[sec]	-	6.0	3.0	6.0	3.0
mean interarrival during bursts[sec]	-	0.6	0.15	0.6	0.15
mean interarrival time between bursts[min]	-	6	36	6	36
mean CPU usage	-	-	-	0.01	0.01
mean RAM usage	-	-	-	0.01	0.01
mean CPU usage (production)	-	-	-	0.02	0.02
mean RAM usage (production)	-	-	-	0.02	0.02
mean execution time[sec]	-	-	-	3400	3400
mean execution time[sec] (production)	-	-	-	27200	27200
Parameter	W-A_l	W-B_l	W-C_l	W-D_l	W-E_l
mean interarrival time[sec]	-	8.0	4.0	8.0	4.0
mean interarrival during bursts[sec]	-	0.8	0.20	0.8	0.20
mean interarrival time between bursts[min]	-	6	36	6	36
mean CPU usage	-	-	-	0.01	0.01
mean RAM usage	-	-	-	0.01	0.01
mean CPU usage (production)	-	-	-	0.02	0.02
mean RAM usage (production)	-	-	-	0.02	0.02
mean execution time[sec]	-	-	-	3400	3400
mean execution time[sec] (production)	-	-	-	27200	27200

10. CONCLUSIONS

This chapter summarizes the contributions of this thesis and presents several promising research directions that can utilize our findings and proposed solutions.

10.1. Summary of Contributions

Main contributions of this thesis can be summarized as follows:

(i) *Network-Aware Task Scheduling:* In this thesis, we propose an online task scheduling scheme which relies on an opportunity cost based approach by providing a single marginal cost value for multidimensional resources. Diverging from the solutions in the literature, our algorithm provides a joint approach by controlling server and network resources simultaneously. The success of proposed scheduler relies on ensuring less tight packing of tasks on servers in order to leave room for bursty arrivals. Thus, our scheduler provides better response time and resource utilization for workloads with bursty arrivals and heterogeneous resource requirements.

(ii) *Resource inefficiencies associated with priority scheduling:* Observations arising from real trace studies help us to better understand system challenges, thereby facilitating design of schedulers to meet new system requirements. By making characterization analysis with real production trace – Google Cluster Trace – we show that significant amount of computing resources are wasted due to unsuccessful executions, mainly priority evictions. Moreover, by demonstrating the reasons for resource inefficiency, we provide a solid background for devising efficient priority schedulers for workloads with varying performance goals.

(iii) *Priority scheduler design, mitigating the impact of evictions:* Motivated by the high complexity of workloads and the significant resource inefficiency in priority-based schedulers, we propose a trace-driven cluster management framework that enables exploring the design space of scheduling policies with a particular focus on the impact of task evictions. The proposed framework models not only a com-

prehensive set of system and workload parameters, i.e., CPU cores, memory capacities, task slots, priorities, CPU/memory demands, but also a general priority-based scheduler. We evaluate the tradeoff between resource inefficiency and task response times of different priorities under different combinations of scheduling policies and system configurations. The results show that certain eviction polices lead to repetitive evictions which create outliers in response time. We identify the underlying reasons of repetitive evictions in order to optimize scheduling. Based on the highlights we obtained, we propose to impose a limit on the number of evictions which works as a regulator for the reschedulings. Thresholding improves response time by eliminating straggler tasks and increasing system availability.

(iv) *Workload-awareness:* Due to high variability in resource demand of priority classes, we propose to optimize task scheduling by introducing workload-awareness in slot configuration and task assignment. We introduce a workload-aware task scheduling scheme consisting of an offline slot configuration and online assignment algorithm. The workload-aware system proactively configures slots according to workload properties and automatically assigns tasks onto slots based on priorities to meet class specific performance objectives. By the introduction of heterogeneous slots, we provide more resources to high resource demanding high priority classes in order to improve their response time. Our proposed algorithm achieves significant improvements on class-based response times by proactively configuring heterogeneous slots according to the characteristics of workloads, and assigning high priority tasks with high resource demands to powerful slots, i.e., equipped with more CPU and memory capacity. We show that the workload-aware system promises significant improvements not only in resource efficiency but also in class-based response time.

(v) *HYBRID Scheduling:* The priority schedulers impose preemptive prioritization with the cost of increased system load and outlier response times of low priorities. Our proposed HYBRID scheduling policy effectively handles both prioritization and performance issues of low priorities while utilizing a combination of preemptive and non-preemptive scheduling by the usage of sticky slots. The effectiveness of HYBRID system lies at providing scheduling guarantee for long waited tasks and limiting the competition of resources between high and low priorities by

utilizing sticky slots.

Moreover, we present our observations and important factors for a greater understanding of efficient system design in data centers. We also introduce some ideas arising from our analysis of various scheduling algorithms that can be leveraged to design for efficiency of systems.

Scheduling is more than task assignment. Although the main role of the scheduler is to find a match between tasks and servers, due to the complexity of recent systems and workloads, the scheduler needs to be adaptive, reactive/responsive and flexible. Popular simplifications (e.g homogenous systems/workloads, exponential interarrival times, exact knowledge of resource requirements) do not really apply to real systems. Therefore, we present a scheduler design which proactively configures system settings based on workload statistics and assigns tasks to slots based on workload characteristics (see Chapter 6). Our proposed design do not require a priori knowledge of resource demands.

Not only unused resources but also unsuccessful executions contribute to significant inefficiencies. Although most of the prior work focus on improving the energy and resource efficiency by consolidation and turning off unused servers, significant amount of computational resources are wasted by unsuccessful executions. Inefficiencies due to unsuccessful executions are overlooked. We find out the main reasons of unsuccessful executions in Chapter 5.

Evictions lead to strong negative feedback loops. Evictions severely increase system load which lead to a significant increase on response times especially for low priority classes. Increased system load in return increases evictions. Therefore, evictions have a severe negative impact on task success rate.

Repetitive evictions lead to outlier response times. Eviction policy not only affects extra load created by the evictions and resubmissions for non-resume systems, but also affects the probability of creating straggler tasks, by repetitively evicting same set of

tasks. Response time of lowest priority class highly depends on the repetitive evictions, even on resume systems, which is decided by the eviction policy. In this thesis, a simple thresholding mechanism is proposed to regulate the rejoins to the central queue hence boosting the performance by eliminating stragglers. More complex mechanisms can be incorporated to handle repetitive evictions.

Optimizing consolidation is vital. For slot-based systems, increasing number of slots improves lowest priority class response time significantly by eliminating priority evictions and long waiting times. On the other hand, high priority classes suffer from decreased CPU allocation due to increased number of co-executing tasks. This tradeoff is extensively investigated in Chapter 7. Cost effectiveness of task migration can be further explored as a part of task consolidation.

Average response time is not a perfect metric. In order to quantify the performance in multi-class systems with varying number of arrivals, average response time (R) may lead to misleading conclusions since it can not capture the effect of priorities and it is highly dependent on the class population. Value based metrics, like the one (V) we proposed, need to be optimized rather than R.

Heterogeneity is the key factor to be considered when designing schedulers. Workload oblivious approaches lead to non-optimal decisions due to incompatible resource requirements and allocations for highly heterogeneous workloads. In order to effectively handle the workload, we offer to integrate workload-awareness in slot configuration and task assignment in Chapter 8 and in Chapter 9. Even though workload is mostly unpredictable, using main observations from trace analytics helps to achieve better response times and resource efficiency by converging to a workload adaptive system. As an extension, more complex learning analyses can be conducted to provide a workload adaptive system.

REFERENCES

1. Barroso, L. A. and U. Hölzle, "The Case for Energy-Proportional Computing", *IEEE Computer*, Vol. 40, No. 12, pp. 33–37, 2007.

2. Verma, A., L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune and J. Wilkes, "Large-scale cluster management at Google with Borg", *Proceedings of the European Conference on Computer Systems (EuroSys)*, pp. 1–17, 2015.

3. Reiss, C., A. Tumanov, G. R. Ganger, R. H. Katz and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis", *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pp. 1–13, 2012.

4. Dean, J. and L. Barroso, "The tail at scale", *Communications of the ACM*, Vol. 56, No. 2, pp. 74–80, 2013.

5. Cho, B., M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts and P. Lin, "Natjam: Design and Evaluation of Eviction Policies for Supporting Priorities and Deadlines in Mapreduce Clusters", *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pp. 1–17, 2013.

6. Schwarzkopf, M., A. Konwinski, M. Abd-El-Malek and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters", *Proceedings of the European Conference on Computer Systems (EuroSys)*, pp. 351–364, 2013.

7. Zaharia, M., D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker and I. Stoica, "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling", *Proceedings of the European Conference on Computer Systems (EuroSys)*, pp. 265–278, 2010.

8. Lin, M., A. Wierman, L. Andrew and E. Thereska, "Dynamic right-sizing for power-proportional data centers", *IEEE International Conference on Computer*

Communications (INFOCOM), pp. 1098–1106, 2011.

9. Wang, K., M. Lin, F. Ciucu, A. Wierman and C. Lin, "Characterizing the impact of the workload on the value of dynamic resizing in data centers", *ACM SIGMETRICS Performance Evaluation*, Vol. 40, No. 1, pp. 405–406, 2012.

10. Chen, G., W. He, J. Liu, S. Nath, L. Rigas, L. Xiao and F. Zhao, "Energy-aware Server Provisioning and Load Dispatching for Connection-intensive Internet Services", *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 337–350, 2008.

11. Gandhi, A., P. Dube, A. Karve, A. Kochut and L. Zhang, "Adaptive, Model-driven Autoscaling for Cloud Applications", *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pp. 57–64, 2014.

12. Delimitrou, C. and C. Kozyrakis, "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters", *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 77–88, 2013.

13. Gandhi, A., P. Dube, A. Kochut and L. Zhang, "Model-Driven Autoscaling for Hadoop Clusters", *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pp. 155–156, 2015.

14. Hindman, B., A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center", *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 295–308, 2011.

15. Ghodsi, A., M. Zaharia, B. Hindman, A. Konwinski, S. Shenker and I. Stoica, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types", *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 323–336, 2011.

16. Isard, M., V. Prabhakaran, J. Currey, U. Wieder, K. Talwar and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters", *Proceedings of the ACM Symposium on Operating Systems Principles (SIGOPS)*, pp. 261–276, 2009.

17. The Apache Software Foundation, *Hadoop MapReduce Next Generation - Fair Scheduler*, `https://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html`, December 2013.

18. The Apache Software Foundation, *Hadoop MapReduce Capacity Scheduler*, `https://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html`, December 2013.

19. Nair, J., K. Jagannathan and A. Wierman, "When heavy-tailed and light-tailed flows compete: The response time tail under generalized max-weight scheduling", *IEEE International Conference on Computer Communications (INFOCOM)*, pp. 2976–2984, 2013.

20. Sleptchenko, A., A. Harten and M. Heijden, "An Exact Solution for the State Probabilities of the Multi-Class, Multi-Server Queue with Preemptive Priorities", *Queueing Systems*, Vol. 50, No. 1, pp. 81–107, 2005.

21. Sleptchenko, A., I. Adan and G. van Houtum, *Joint queue length distribution of multi-class, single-server queues with preemptive priorities*, Eurandom, 2004.

22. Zhang, Q., M. F. Zhani, S. Zhang, Q. Zhu, R. Boutaba and J. L. Hellerstein, "Dynamic Energy-aware Capacity Provisioning for Cloud Computing Environments", *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pp. 145–154, 2012.

23. Zhang, Q., M. Zhani, R. Boutaba and J. Hellerstein, "Harmony: Dynamic Heterogeneity-Aware Resource Provisioning in the Cloud", *Proceedings of the Distributed Computing Systems (ICDCS)*, pp. 510–519, 2013.

24. Harchol-Balter, M., T. Osogami, A. Scheller-Wolf and A. Wierman, "Multi-Server Queueing Systems with Multiple Priority Classes", *Queueing Systems Theory and Application*, Vol. 51, pp. 331–360, 2005.

25. Jelenkovic, P. R. and E. D. Skiani, "Is Sharing with Retransmissions Causing Instabilities?", *ACM SIGMETRICS Performance Evaluation*, Vol. 42, No. 1, pp. 167–179, 2014.

26. Amazon Web Services, *Assign Priority Levels to Tasks*, `https://aws.amazon.com/about-aws/whats-new/2014/12/17/assign-priority-levels-to-tasks-in-amazon-swf-workflows/`, December 2014.

27. *The Next Generation of Apache Hadoop MapReduce*, `http://hadoop.apache.org/`, December 2014.

28. Sakellari, G. and G. Loukas, "A survey of mathematical models, simulation approaches and testbeds used for research in cloud computing", *Simulation Modelling Practice and Theory*, Vol. 39, pp. 92 – 103, 2013.

29. Cheng, L., Q. Zhang and R. Boutaba, "Mitigating the negative impact of preemption on heterogeneous MapReduce workloads", *International Conference on Network and Service Management (CNSM)*, pp. 1–9, 2011.

30. Ananthanarayanan, G., C. Douglas, R. Ramakrishnan, S. Rao and I. Stoica, "True Elasticity in Multi-tenant Data-intensive Compute Clusters", *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pp. 1–7, 2012.

31. Di, S., D. Kondo and C. Franck, "Characterizing Cloud Applications on a Google Data Center", *Proceedings of the International Conference on Parallel Processing (ICPP)*, pp. 468–473, 2013.

32. Chen, Y., S. Alspaugh, D. Borthakur and R. Katz, "Energy Efficiency for Large-scale MapReduce Workloads with Significant Interactive Analysis", *Proceedings*

of the European Conference on Computer Systems (EuroSys), pp. 43–56, 2012.

33. Wilkes, J., *More Google cluster data*, `https://code.google.com/p/googleclusterdata/wiki/ClusterData2011_1`, March 2011.

34. Mishra, A. K., J. L. Hellerstein, W. Cirne and C. R. Das, "Towards characterizing cloud backend workloads: insights from Google compute clusters", *ACM SIGMETRICS Performance Evaluation*, Vol. 37, No. 1, pp. 34–41, 2010.

35. Di, S., D. Kondo and W. Cirne, "Characterization and comparison of cloud versus Grid workloads", *Proceedings of the International Conference on Cluster Computing (CLUSTER)*, pp. 230–238, 2012.

36. Liu, Z., Y. Chen, C. Bash, A. Wierman, D. Gmach, Z. Wang, M. Marwah and C. Hyser, "Renewable and Cooling Aware Workload Management for Sustainable Data Centers", *ACM SIGMETRICS Performance Evaluation*, Vol. 40, No. 1, pp. 175–186, 2012.

37. Abdul-Rahman, O. A. and K. Aida, "Towards understanding the usage behavior of Google cloud users: the mice and elephants phenomenon", *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 272–277, 2014.

38. Jiang, W., C. Hu, Y. Zhou and A. Kanevsky, "Are Disks the Dominant Contributor for Storage Failures?: A Comprehensive Study of Storage Subsystem Failure Characteristics", *ACM Transactions on Storage*, Vol. 4, No. 3, pp. 1–25, 2008.

39. Bairavasundaram, L. N., A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson and B. Schroeder, "An Analysis of Data Corruption in the Storage Stack", *ACM Transactions on Storage*, Vol. 4, No. 3, pp. 1–28, 2008.

40. Gill, P., N. Jain and N. Nagappan, "Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications", *ACM SIGCOMM Computer*

Communication Review, pp. 350–361, 2011.

41. Potharaju, R. and N. Jain, "When the Network Crumbles: An Empirical Study of Cloud Network Failures and Their Impact on Services", *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pp. 1–17, 2013.

42. Benson, T., A. Akella and D. A. Maltz, "Network traffic characteristics of data centers in the wild", *ACM SIGCOMM Computer Communication Review*, pp. 267–280, 2010.

43. Turner, D., K. Levchenko, A. C. Snoeren and S. Savage, "California Fault Lines: Understanding the Causes and Impact of Network Failures", *ACM SIGCOMM Computer Communication Review*, pp. 315–326, 2010.

44. Birke, R., I. Giurgiu, L. Chen, D. Wiesmann and T. Engbersen, "Failure Analysis of Virtual and Physical Machines: Patterns, Causes and Characteristics", *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12, 2014.

45. Vishwanath, K. V. and N. Nagappan, "Characterizing Cloud Computing Hardware Reliability", *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pp. 193–204, 2010.

46. Sahoo, R., M. Squillante, A. Sivasubramaniam and Y. Zhang, "Failure data analysis of a large-scale heterogeneous server environment", *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pp. 772–781, 2004.

47. Chen, Y., A. S. Ganapathi, R. Griffith and R. H. Katz, *Analysis and Lessons from a Publicly Available Google Cluster Trace*, `http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-95.html`, May 2010.

48. Sharma, B., R. Prabhakar, S. Lim, M. Kandemir and C. Das, "MROrchestra-

tor: A Fine-Grained Resource Orchestration Framework for MapReduce Clusters", *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, pp. 1–8, 2012.

49. Grandl, R., G. Ananthanarayanan, S. Kandula, S. Rao and A. Akella, "Multi-resource Packing for Cluster Schedulers", *ACM SIGCOMM Computer Communication Review*, pp. 455–466, 2014.

50. Zhang, Q., M. Zhani, R. Boutaba and J. Hellerstein, "Dynamic Heterogeneity-Aware Resource Provisioning in the Cloud", *IEEE Transactions on Cloud Computing*, Vol. 2, No. 1, pp. 14–28, 2014.

51. Kavulya, S., J. Tan, R. Gandhi and P. Narasimhan, "An analysis of traces from a production mapreduce cluster", *Proceedings of the IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 94–103, 2010.

52. Sharma, B., V. Chudnovsky, J. L. Hellerstein, R. Rifaat and C. R. Das, "Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters", *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pp. 1–14, 2011.

53. *Apache Hadoop*, `http://hadoop.apache.org/`, December 2014.

54. Wang, J., Y. Yao, Y. Mao, B. Sheng and N. Mi, "FRESH: Fair and Efficient Slot Configuration and Scheduling for Hadoop Clusters", *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, pp. 761–768, 2014.

55. Tang, S., B.-S. Lee and B. He, "DynamicMR: A Dynamic Slot Allocation Optimization Framework for MapReduce Clusters", *IEEE Transactions on Cloud Computing*, Vol. 2, No. 3, 2014.

56. Polo, J., C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres and E. Ayguadé, "Resource-aware Adaptive Scheduling for Mapreduce Clusters", *Proceedings of the ACM/IFIP/USENIX Middleware conference*, pp. 187–207, 2011.

57. Tan, J., X. Meng and L. Zhang, "Coupling task progress for MapReduce resource-aware scheduling", *IEEE International Conference on Computer Communications (INFOCOM)*, pp. 1618–1626, 2013.

58. Nelson, M., B.-H. Lim, G. Hutchins *et al.*, "Fast Transparent Migration for Virtual Machines.", *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 391–394, 2005.

59. *VMware infrastructure architecture overview*, `http://www.vmware.com/pdf/vi_architecture_wp.pdf`, May 2014.

60. Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield, "Xen and the art of virtualization", *Proceedings of the ACM symposium on Operating systems principles (SOPS)*, pp. 164–177, 2003.

61. Lu, L., X. Zhu, R. Griffith, P. Padala, A. Parikh, P. Shah and E. Smirni, "Application-driven dynamic vertical scaling of virtual machines in resource pools", *Proceedings of the IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–9, 2014.

62. Clark, C., K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt and A. Warfield, "Live Migration of Virtual Machines", *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 273–286, 2005.

63. Li, B., J. Li, J. Huai, T. Wo, Q. Li and L. Zhong, "EnaCloud: An Energy-Saving Application Live Placement Approach for Cloud Computing Environments", *Proceedings of the IEEE International Conference on Cloud Computing*, pp. 17–24, 2009.

64. Buyya, R., A. Beloglazov and J. Abawajy, "Energy-Efficient Management of Data Center Resources for Cloud Computing : A Vision, Architectural Elements, and Open Challenges", *Proceedings of the International Conference on Parallel and*

Distributed Processing Techniques and Applications (PDPTA), 2010.

65. Calheiros, R. N., R. Ranjan, A. Beloglazov, C. A. F. De Rose and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms", *Software: Practice and Experience*, Vol. 41, No. 1, pp. 23–50, 2011.

66. Srikantaiah, S., A. Kansal and F. Zhao, "Energy aware consolidation for cloud computing", *Proceedings of the conference on Power aware computing and systems (HotPower)*, 2008.

67. Govindan, S., J. Liu, A. Kansal and A. Sivasubramaniam, "Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines", *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pp. 1–14, 2011.

68. Dean, J. and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", *Communications of the ACM*, Vol. 51, pp. 107–113, 2008.

69. Ananthanarayanan, G., S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha and E. Harris, "Reining in the Outliers in Map-reduce Clusters Using Mantri", *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 1–16, 2010.

70. Zaharia, M., A. Konwinski, A. D. Joseph, R. Katz and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments", *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 29–42, 2008.

71. Cirne, W., F. Brasileiro, D. Paranhos, L. F. W. Góes and W. Voorsluys, "On the Efficacy, Efficiency and Emergent Behavior of Task Replication in Large Distributed Systems", *Parallel Computing*, Vol. 33, No. 3, pp. 213–234, 2007.

72. Merchant, A., M. Uysal, P. Padala, X. Zhu, S. Singhal and K. Shin, "Maestro: Quality-of-Service in Large Disk Arrays", *Proceedings of the ACM international conference on Autonomic computing (ICAC)*, pp. 245–254, 2011.

73. *Google Inc., Efficient Data Center Summit*, `http://www.google.com/corporate/green/datacenters/summit.html`, April 2009.

74. Pouwelse, J., K. Langendoen and H. Sips, "Energy priority scheduling for variable voltage processors", *Proceedings of the international symposium on Low power electronics and design (ISLPED)*, pp. 28–33, 2001.

75. Benini, L., A. Bogliolo and G. De Micheli, "A survey of design techniques for system-level dynamic power management", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 8, No. 3, pp. 299 –316, 2000.

76. Kliazovich, D., P. Bouvry and S. U. Khan, "GreenCloud: a packet-level simulator of energy-aware cloud computing data centers", *The Journal of Supercomputing*, Vol. 62, No. 3, pp. 1263–1283, 2010.

77. Chekuri, C. and S. Khanna, "On Multidimensional Packing Problems", *SIAM Journal on Computing*, Vol. 33, No. 4, pp. 837–851, 2004.

78. Bansal, N., J. R. Correa, C. Kenyon and M. Sviridenko, "Bin Packing in Multiple Dimensions: Inapproximability Results and Approximation Schemes", *Mathematics of Operations Research*, Vol. 31, No. 1, pp. 31–49, 2006.

79. Woeginger, G. J., "There is no asymptotic PTAS for two-dimensional vector packing", *Information Processing Letters*, Vol. 64, No. 6, pp. 293 – 297, 1997.

80. Panigrahy, R., K. Talwar, L. Uyeda and U. Wieder, *Heuristics for Vector Bin Packing*, `http://research.microsoft.com/apps/pubs/default.aspx?id=147927`, May 2011.

81. Dean, J. and S. Ghemawat, "MapReduce: Simplified Data Processing on Large

Clusters", *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 10–10, 2004.

82. Meisner, D., B. T. Gold and T. F. Wenisch, "PowerNap: Eliminating Server Idle Power", *ACM SIGARCH Computer Architecture News*, Vol. 37, No. 1, pp. 205–216, 2009.

83. Gandhi, A., M. Harchol-Balter, R. Das and C. Lefurgy, "Optimal power allocation in server farms", *ACM SIGMETRICS Performance Evaluation*, Vol. 37, No. 1, pp. 157–168, 2009.

84. Gandhi, A., V. Gupta, M. Harchol-balter and M. A. Kozuch, "Optimality Analysis of Energy-Performance Trade-off for Server Farm Management", *ACM SIGMETRICS Performance Evaluation*, Vol. 67, No. 11, pp. 1–23, 2010.

85. Heller, B., S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee and N. McKeown, "ElasticTree: saving energy in data center networks", *Proceedings of the USENIX conference on Networked systems design and implementation (NSDI)*, 2010.

86. Fang, W., X. Liang, S. Li, L. Chiaraviglio and N. Xiong, "VMPlanner: Optimizing virtual machine placement and traffic flow routing to reduce network power costs in cloud data centers", *Computer Networks*, Vol. 57, No. 1, pp. 179 – 196, 2013.

87. Huang, D., D. Yang, H. Zhang and L. Wu, "Energy-aware virtual machine placement in data centers", *Proceedings of the IEEE Global Communications Conference (GLOBECOM),*, pp. 3243–3249, 2012.

88. Wang, L., F. Zhang, A. V. Vasilakos, C. Hou and Z. Liu, "Joint Virtual Machine Assignment and Traffic Engineering for Green Data Center Networks", *Proceedings of the GreenMetrics Workshop*, 2013.

89. Reiss, C., A. Tumanov, G. R. Ganger, R. H. Katz and M. A. Kozuch, *Towards*

understanding heterogeneous clouds at scale:Google trace analysis, April 2012.

90. Mahadevan, P., P. Sharma, S. Banerjee and P. Ranganathan, "A Power Benchmarking Framework for Network Devices", *Proceedings of the International IFIP Networking Conference (NETWORKING)*, pp. 795–808, 2009.

91. Wobker, L. J., *Power Consumption in High-End Routing Systems*, `http://www.nanog.org/meetings/nanog54/presentations/Wednesday/Wobker.pdf`, March 2014.

92. Reiss, C., J. Wilkes and J. L. Hellerstein, *Google cluster-usage traces: format + schema*, `code.google.com/p/googleclusterdata/wiki/TraceVersion2`, May 2012.

93. Spicuglia, S., L. Chen and W. Binder, "Join the Best Queue: Reducing Performance Variability in Heterogeneous Systems", *Proceedings of the International Conference on Cloud Computing (CloudCom)*, pp. 139–146, 2013.

94. Gandhi, A., M. Harchol-Balter, R. Raghunathan and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers", *ACM Transactions on Computer Systems (TOCS)*, Vol. 30, No. 4, pp. 1–26, 2012.

95. Sharma, B., T. Wood and C. R. Das, "HybridMR: A Hierarchical MapReduce Scheduler for Hybrid Data Centers", *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2013.

96. Gandhi, A., M. Harchol-Balter and M. A. Kozuch, "The Case for Sleep States in Servers", *Proceedings of the conference on Power aware computing and systems (HotPower)*, pp. 1–5, 2011.

97. Warneke, D. and C. Leng, "A Case for Dynamic Memory Partitioning in Data Centers", *Proceedings of the Workshop on Data Analytics in the Cloud*, pp. 41–45, 2013.

98. Gandhi, A., M. Harchol-Balter and M. A. Kozuch, "Are Sleep States Effective in Data Centers?", *Proceedings of the International Green Computing Conference (IGCC)*, pp. 1–10, 2012.

99. *ENERGY STAR Computer Server Qualified Product List*, `www.energystar.gov/ia/products/prod_lists/enterprise_servers_prod_list.xls`, May 2014.

100. Baliga, J., R. Ayre, K. Hinton and R. Tucker, "Green Cloud Computing: Balancing Energy in Processing, Storage, and Transport", *Proceedings of the IEEE*, Vol. 99, No. 1, pp. 149–167, 2011.

101. Beloglazov, A., J. Abawajy and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing", *Future Generation Computer Systems*, Vol. 28, No. 5, pp. 755 – 768, 2012.

102. Ren, S., Y. He, S. Elnikety and K. S. McKinley, "Exploiting Processor Heterogeneity in Interactive Services", *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pp. 45–58, 2013.

103. Kvanli, A., R. Pavur and K. Keeling, *Concise Managerial Statistics*, Cengage Learning, pp. 80-83, 2005.

104. Hamdaoui, M. and P. Ramanathan, "A dynamic priority assignment technique for streams with (m, k)-firm deadlines", *IEEE Transactions on Computers*, Vol. 44, No. 12, pp. 1443–1451, 1995.

105. Herodotou, H., F. Dong and S. Babu, "No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics", *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pp. 1–14, 2011.