UC Davis
UC Davis Electronic Theses and Dissertations

Title
Applied Machine Learning for Resource Provisioning of Data-Intensive Applications on Scale-Out Platforms and Its Security Challenges

Permalink
https://escholarship.org/uc/item/4vg9t5tr

Author
Mohammadi Makrani, Hosein

Publication Date
2021

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Applied Machine Learning for Resource Provisioning of Data-Intensive Applications on Scale-Out Platforms and Its Security Challenges

By

HOSEIN MOHAMMADI MAKRANI

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

____

Houman Homayoun, Chair

____

John D. Owens

____

Venkatesh Akella

Committee in Charge

2021

To my beloved wife Najmeh, and our parents

Abstract

Applied Machine Learning for Resource Provisioning of Data-Intensive Applications on Scale-Out Platforms and Its Security Challenges

The processing of data-intensive applications is a challenging and time-consuming task that often requires massive infrastructure to ensure fast data analysis. One of the most powerful scale-out infrastructures to perform massive computation (e.g. big data analytics) and eliminate the need to maintain high-end expensive computing resources at the user side is the cloud. The performance and the cost of such infrastructure depend on the overall server configuration, such as processor, memory, network, and storage configurations. In addition to the cost of owning or maintaining the hardware, the heterogeneity in the server configuration further expands the selection space, leading to non-convergence. The challenge is further exacerbated by the dependency of the application's performance on the underlying hardware.

Despite an increasing interest in resource provisioning, little works have been done in developing accurate and practical models to proactively predict the performance of data-intensive applications corresponding to the server configuration and provision an optimal configuration online. The key challenges of current solutions are uncertainty in predictions, cost of training, generalizability from benchmark datasets to real-world systems datasets, and interpretability of the model.

In this dissertation, through a comprehensive real-system empirical analysis of performance, we address these challenges by introducing a proactive machine-learning-based methodology for resource provisioning. We first characterize diverse types of data-intensive workloads across different types of server architectures. The characterization aids in accurately capture applications' behavior and train a model for the prediction of their performance. Then, we build a set of cross-platform performance models for applications. Based on the developed

predictive model, we use optimization techniques to distinguish close-to-optimal configurations in order to reach the performance goal.

On the other hand, in recent literature, researchers substantiated that the machine learning-based models bring new security challenges such as adversarial machine learning attacks. In this dissertation, we investigate what could be the target of adversarial machine learning in the cloud domain and how much the risk of this new thread is real. To the best of our knowledge, we are the first group looking into this domain of research as no report has been found on the adversarial attacks on resource provisioning systems (RPS) of the cloud. Our investigation shows that adversarial machine learning can be used for co-locating the adversary Virtual Machines (VM) with the victim VM to attack to its performance. Moreover, we show that the attacker can fool the RPS to evade the detection and migration performed by RPS.

Acknowledgments

Many people are responsible for supporting me in completing this dissertation. First and foremost, I am extremely grateful to my Ph.D. advisor, Dr. Houman Homayoun. He offered me the great opportunity of working on a topic I am passionate about. I want to thank him for his advice not only on research but career paths and life in general. Without his full support, non of my Ph.D. works were achievable. I want to say thank you to my other collaborator, Hossein Sayadi for his efforts and collaborations in bringing our projects into success. I want to thank my peers from ASEEC Lab, including but not limited to Han Wang, Katayoun Neshatpour, Maria Malik, and Gaurav Kolhe that gave me a memorable Ph.D. journey.

I would like to thank Jayshree Sarma, who has been an invaluable help whenever I ran into a problem with using the Argo servers. Her assistance has allowed several of my research papers to meet the deadlines.

Many thanks to my internship mentors: Dr. Malek Ben Salem, and Dr. Hooman Torabi Parizi, who helped me with new ideas and technical guidance on real-world problems.

I am also thankful to all the faculty members of George Mason University, and especially Dr. Avesata Sasan, Dr. Setareh Rafatirad, and Dr. Sai Manoj P D for their advice, feedback, and inspiration over the years that I studied at George Mason University. I would like to express my appreciation for the helpful comments and guidance from my dissertation committee members at the University of California, Davis: Prof. John D. Owens, and Prof. Venkatesh Akella. In addition to them, I want to thank Prof. Zhi Ding and Dr. Mohammad Sadoghi from my qualifying exam committee for their advice.

Contents

Contents vi

List of Figures viii

List of Tables x

1 Introduction 1
- 1.1 Challenge of Diversity . . . 2
- 1.2 Security Threats in Cloud Rooted from ML in RPS . . . 4
- 1.3 Overview of the dissertation . . . 6
- 1.4 Organization . . . 11

2 State of the Arts on ML-Assisted Resource Provisioning Systems 13
- 2.1 Characterization . . . 13
- 2.2 Resource Provisioning Systems . . . 15

3 Performance Analysis of Data-Intensive Applications 18
- 3.1 Introduction . . . 19
- 3.2 Experimental Setup . . . 21
- 3.3 Results . . . 28
- 3.4 Conclusion . . . 43

4 Memory Navigator for Modern Hardware in a Scale-out Environment 45
- 4.1 Introduction . . . 45
- 4.2 Experimental Setup . . . 47
- 4.3 Characterization and Results . . . 48
- 4.4 Performance and Cost Analysis . . . 56
- 4.5 Memory Navigator (MeNa) . . . 60
- 4.6 Conclusion . . . 67

5 Energy-Aware and NN-based RPS for IMC Application 69
- 5.1 Introduction . . . 69

5.2 Background and Motivation 70
5.3 E-Net Methodology 73
5.4 Implementation 81
5.5 Results and Evaluation 86
5.6 Conclusion 94

6 Proactive and ML-based RPS for Data-Intensive Applications 95
6.1 Contribution and Novelty of ProMLB 96
6.2 ProMLB 97
6.3 Implementation 108
6.4 Evaluation of ProMLB 110
6.5 Conclusion 116

7 Railroading of Resource Sharing-based Attacks on the Cloud 118
7.1 Background and Threat Model 119
7.2 Cloak & Co-locate 123
7.3 Reverse engineering of RPS 126
7.4 Cloak Generator 129
7.5 Evaluation 133
7.6 Conclusions 141

8 Future Directions and Conclusion 143
8.1 Avenues for Future Directions in ML Research 144
8.2 Concluding Remarks 145

Bibliography 147

List of Figures

1.1 Impact of resource provisioning on the cloud's aspects 3
1.2 Microarchitectural break-down of workloads for different phases 4
1.3 MLbased resource provisioning system 7

3.1 Effect of memory channel on the execution time (Normalized to 4CH) 30
3.2 Impact of memory channel on bandwidth usage 31
3.3 Effect of memory frequency on the execution time (Normalized to 2400MHz) . . 32
3.4 Impact of memory capacity per node on performance 33
3.5 K-means memory usage on various frameworks 34
3.6 Average results of input size's effect on memory behavior 35
3.7 Impact of CPU frequency on the execution time 36
3.8 C0 residency of processor 37
3.9 LLC and L2 hit rate 37
3.10 Effect of memory and storage configuration on the performance 38
3.11 Average memory bandwidth utilization 39
3.12 Effect of core count on the performance 40
3.13 Lambda value for different request size and storage type 41
3.14 Average error of optimum core count prediction 41
3.15 DRAM power consumption 41
3.16 Average normalized EDP (Normalized to 4CH, 2400MHz, 2.6GHz) 42

4.1 Memory sensitivity analysis 50
4.2 Bandwidth utilization 52
4.3 Bandwidth usage 52
4.4 Speedup by memory 53
4.5 Speedup by CPU frequency 53
4.6 Workloads' microarchitectural behavior 54
4.7 Disk access of Big Data applications 55
4.8 MeNa methodology overview 61
4.9 MeNa classifier (Neural Network) 61
4.10 Average error for different class of applications and Budget 63
4.11 Average Performance/Cost correspond to server parameters 66

4.12 Effect of memory parameters on Performance/Cost 66
4.13 Average Performance/Cost correspond to Budget 67

5.1 Example of application behavior and phase change 71
5.2 Average phase distribution of Spark workloads 72
5.3 E-Net overview 73
5.4 Overview of E-Net cluster 83
5.5 Overview of benchmarking and training 85
5.6 Error rate of ML-based predictors 88
5.7 Impact of the number of windows on the accuracy of TSNNs 88
5.8 Comparison of Optimization techniques (BF as a baseline) 89
5.10 Impact of running multiple jobs on EDP 92
5.11 Correlation between the server type and average utilization 92

6.1 ProMLB overview 98
6.2 Block diagram of ProMLB 99
6.3 Performance model generated for a workload 103
6.4 Migration time 106
6.5 Dataset division for the training and testing 109
6.6 Impact of window's size and number of windows on accuracy of predictors . . . 112
6.7 Overall accuracy of predictors 113
6.8 Scatter plots of prediction values versus 114

7.1 Overview of Cloak & Co-locate 125
7.2 Reverse engineering process 128
7.3 Coverage of application's characteristic 130
7.4 Fast Gradient Sign Method 131
7.5 Activation of FTG 132
7.6 Overall trace using FTG, adversary kernel trace, and the target 132
7.7 Impact of monitoring period on the chance of evasion from migration 134
7.8 Impact of similarity with the target trace on migration chance 135
7.9 Impact of perturbation on migration chance 136
7.10 Impact of perturbation on attack success rate 136
7.11 Average of attacks' success rate when multiple VMs are running in the host . . 137
7.12 Breakdown of attacks' success or failure on a general purpose server 138
7.13 Attacks' success rate and CPU utilization (Util) with isolation techniques 139

List of Tables

2.1 Comparison of state of the arts . . . 16

3.1 Studied workloads . . . 23
3.2 Hardware Platform . . . 25
3.3 HDFS block size tuning . . . 28

4.1 Big Data Workloads . . . 47
4.2 Memory modules' part numbers . . . 48
4.3 IBM SoftLayer bare metal servers . . . 48
4.4 Hardware Platform . . . 49
4.5 Values of processor cost's formula . . . 58
4.6 Performance gain by increasing core count . . . 60
4.7 Performance gain by increasing core frequency . . . 60
4.8 Performance gain by memory frequency and channel . . . 60
4.9 MeNa Validation . . . 63
4.10 Applications' features . . . 64
4.11 Configurations selected for Spark Nweight . . . 64
4.12 Configurations selected for Hadoop Terasort . . . 64
4.13 Configurations selected for Hadoop Scan . . . 65
4.14 Configurations selected for Spark sort . . . 65

5.1 Detailed information of local cluster . . . 83
5.2 Information of scheduling decisions (Average results for each Virtual Machine) . 87

6.1 Average prediction time of each predictor. . . . 111

7.1 Success Rate (SR) of distributed attacks . . . 134
7.2 Attacks' average success rate . . . 141

Chapter 1

Introduction

The information age brought along an explosion of big data from multiple sources in every aspect of our lives [142]. Recent trends suggest that for efficient processing of data-intensive applications, there is a strong demand to find effective solutions for improved data storage, real-time processing, and energy-efficient processing [18]. In response, distributed platforms have emerged as a solution to address these challenges [18]. For a such environment, several frameworks such as Hadoop [42], Spark [44], Flink [41], and Tez [45] have been developed in recent years.

In-Memory cluster Computing (IMC) frameworks such as Spark and Flink have become increasingly important and popular as they achieve multifold speedup over traditional On-Disk Cluster Computing (ODC) frameworks for iterative and interactive applications [31]. However, a key challenge for IMC is that its performance is highly sensitive to the underlying processing and memory configuration, thus requiring the developers to navigate through a large design space to determine a configuration that leads to the optimal performance [130].

At the same time, the advancements of hardware architecture designs lead to datacenters with diverse hardware systems. This hardware diversification at different levels is marking the beginning of an era of *super-heterogeneous datacenters*. From the application perspective, as different applications have different characteristics, one architectural configuration

fits all does not provide the best performance and energy efficiency for every application. This calls for developing an effective strategy to determine the most suitable architecture (resource provisioning) in a heterogeneous datacenter for a given application to deal with the fast-growing data using the existing resources more efficiently. This dissertation aims at developing automatic resource management mechanisms for the scale-out platforms that enable distributed systems to achieve faster and predictable performance while reducing the cost and energy.

1.1 Challenge of Diversity

Virtualization is a process of resource sharing and isolation of underlying hardware to increase computer resource utilization, efficiency, and scalability. Therefore, the cloud service providers offer a wide range of cloud configuration choices such as VM instances with a variety of CPUs, memory, disk, and network configurations and also customized VMs for analytics applications.

The super-heterogeneous datacenter makes determining the best cloud configuration for a given application by brute-force search expensive and exhaustive. Choosing the right cloud configuration is essential, as a non-optimal configuration results in more cost for the same performance target as different analytic jobs have diverse behaviors and resource requirements. As Figure 1.1 illustrates, an efficient resource provisioning impacts three different aspects of the cloud. It fulfills Service-level agreements (SLA) and meets cloud customers' requirements. It guarantees cloud obligations to its users. It also prevents resource waste, thereby reducing energy consumption and the operational cost. The reduction of energy consumption leads to a decrease in carbon emission, which facilitates green computing. Hence, energy-aware resource provisioning is also important for reducing cost and for increasing revenue that improves the profit of cloud providers [120].

A more challenging problem is that the behavior and resource requirements of applications running on the cloud vary during different phases of execution. Each application faces

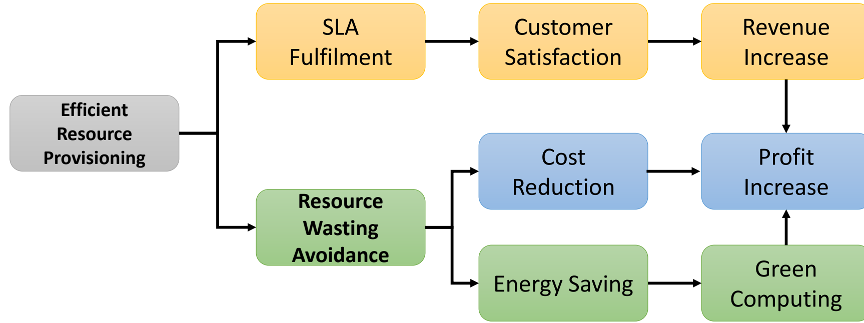

Figure 1.1: Impact of resource provisioning on the cloud's aspects

various phases of execution, each with different memory and processing requirements. Based on the Top-down methodology [128] , three major phases can be identified in an application namely I/O bound phase, memory bound phase, and compute bound phase. These phases are different in terms of their microarchitectural behavior, therefore requiring different processing and memory resources for performance and energy-efficiency optimization. For instance, compute-bound phase requires more cores, higher core frequency, and higher DRAM bandwidth.

Figure 1.2 illustrates the microarchitectural differences between those three phases. The micro-op (µop) queue of an out-of-order processor is used to abstract the microarchitectural behavior. The op queue is classified in four broad categories: Retiring, Front-end bound, Bad speculation, Back-end bound. Out of these categories, only the Retiring is classified as "useful work" while the rest prevents the workload from utilizing the full core bandwidth. In addition to µop queues, C0 (active state residency of processor) is a metric that can be used to differentiate among phases. As the figure shows, the main difference between memory bound and I/O bound is C0 residency. This can be explained as follows: in I/O intensive phase, the core is waiting for I/O, hence the core changes its state to save power. Therefore, C0 residency drops.

There are thousands of applications running at the same time in a cloud and each requiring different processing and memory resources to be allocated at different phases of runtime. It is, therefore, necessary that resource management system identifies those phases at run-

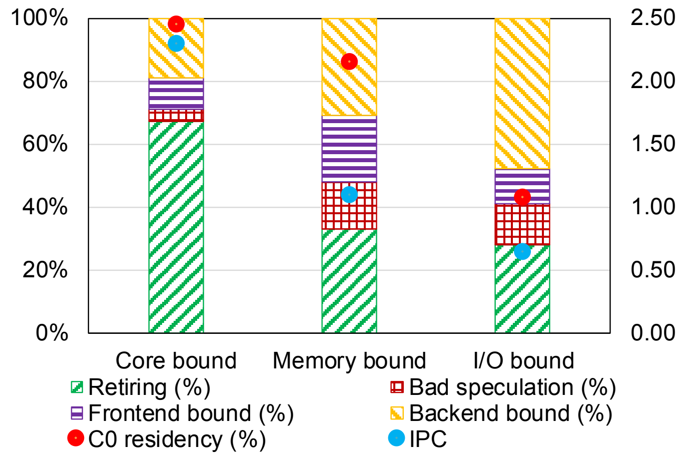

Figure 1.2: Microarchitectural break-down of workloads for different phases

time, to be able to allocate resources accordingly. Hence, this makes existing traditional reactive resource allocation methodologies achieve a sub-optimal performance gain and hard to apply effectively to emerging cloud computing services.

1.2 Security Threats in Cloud Rooted from ML in RPS

Although maximizing utilization, i.e., sharing of resources, is key to achieving cost-efficiency in the cloud, it also opens the door for security and privacy vulnerabilities. In particular, these resources will be shared among different users, due to the multi-tenancy capability of hosts in the cloud, which facilitate a platform for performing a wide range of resource sharing-based attacks, including transient execution attacks [17], rowhammer attacks [123], distributed side-channel attacks [65] and distributed denial of service attacks (DDoS) [34], data leakage exploitation [139, 116], and attacks that pinpoint target VMs in a cloud system [125]. Mounting such attacks is trivial once the attacker is co-located with the victim. Therefore, the biggest challenge of attacks that exploit resource sharing in cloud environ-

ments is co-location [138].

Unfortunately, RPSs can become a blind spot and vulnerability that can be exploited to solve the co-location challenge of resource sharing-based attacks. In particular, adversarial attacks against machine learning models can be adopted to force RPSs to co-locate the attacker with the victim. A plethora of works on adversarial attacks exists, focusing specifically on computer vision applications [15, 30, 60, 90]. These attacks work by adding specially crafted perturbations to the input data, i.e., an image, of machine learning models to manipulate their outcome. However, pixels of an image can be easily manipulated independently without changing the appearance of the image since images have high entropy. In contrast, adding adversarial perturbations to attack programs have different challenges since the attacker needs to ensure that the adversarial perturbations do not alter the malicious payload.

We urge that current RPSs can be exploited to facilitate a wide range of attacks by solving the co-location challenge in the cloud and also highlight a serious need for new techniques to be invented that guarantee security. Specifically, although there is a large number of defenses classes that were developed against computer vision-based adversarial attacks, these defenses are limited in defending against such attacks to RPSs. In particular, these defenses assume that the attacker has a budget, i.e., the maximum amount of perturbation that can be added to an image without changing it's content. This is important in the computer vision domain, since the goal of adversaries is to perturb an image to fool a specific machine learning classifier, but can still be classified correctly by a human. However, for programs perturbations, there is no such budget/constraint, allowing the attacker to have an unlimited degree of freedom to add perturbations without risking increasing the possibility of being detected.

After co-locating the adversarial VMs with the targeted victim, attackers face two more challenges, namely detection and migration. Specifically, the RPS job does not stop after the instance initialization phase, i.e., initial deployment of a VM on a suitable host. It has been shown that periodic monitoring after the initial deployment can be unitized to improve

both security and performance. For security, the trace information can be used to detect attacks based on computational anomaly [22]. For performance, the trace can be utilized to detect performance degradation due to resource contention or behavioral change of the running application and migrate the VM to a different host. However, attacks can evolve to bypass such detection as well as avoiding VM migration.

1.3 Overview of the dissertation

Aforementioned challenges have motivated us to devise a new resource management methodology in the scale-out environment. A resource provisioning system (RPS) facilitates various services including resource efficiency, security, fault tolerance, and monitoring to achieve the performance goals while maximizing the utilization of available resources. To this end, our contributions focus on using machine learning solutions to overcome the challenge of application diversity and heterogeneity of resources.

Several machine learning based resource provisioning systems have been proposed for cloud systems in literature that we will discuss their details in the chapter 2, the state of the art of RPSs. As we mentioned, the RPS attempts to meet the user performance requirements and provider efficiency in terms of multiple aspects such as load balancing among servers, minimum number of active hosts, and least response time, to avoid service-level agreement (SLA) violations in the cloud platform. Hence, RPSs or schedulers to fulfil their objectives must have two main tasks [78]: 1) Instance initialization and 2) Periodic monitoring of applications.

During the instance initialization stage, when an instance is created and submitted to scheduler, the scheduler profiles the application and based on the application's behavior determines the resources required for meeting its SLA. Machine learning can be used in this stage to identify application's characteristics and determine its basic requirements. After that, scheduler allocates the instance to a host in the infrastructure.

During the periodic monitoring stage, scheduler monitors application's behavior to guar-

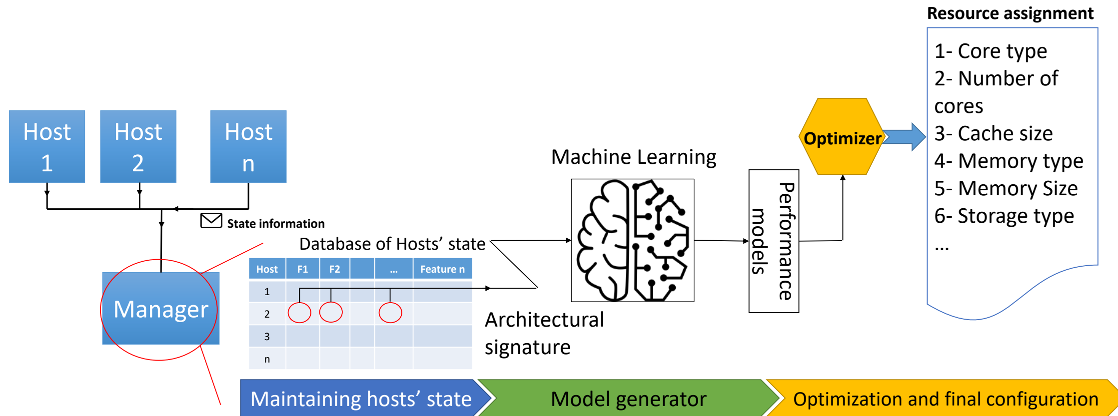

Figure 1.3: MLbased resource provisioning system

antee the SLA all the time. In a case that application's behavior changes, the scheduler attempts to reschedule and migrate the instance to a new host to provide required resources to meet the SLA agreement. In this stage, machine learning can be leveraged to first detect the behavior change, and secondly to model the performance, cost, or even the energy of application for determining the best instance that can cope with the change of application's requirements.

Figure 1.3 shows how a general ML-based RPS works. First, the system monitors the application and extracts its micro-architectural and system level information. Then based on the current behavior and server configuration, it may predict the performance of application to make sure that performance of application will not be degraded. If the RPS identifies a performance degradation, then by leveraging optimization techniques, it determines another suitable configuration and host for the application. We propose a new approach that leverages the knowledge on application behavior the system accumulates through data collection over time. By applying data mining principles to these datasets in a mindful manner, we significantly improve both the quality and practicality of large-scale resource provisioning.

In the context of the problem domains described in the previous sections, below we outline the systems we designed, implemented, and evaluated.

A Comprehensive Performance Analysis of Data-Intensive Workloads on Server Class Architecture

Given the large size and heterogeneity of the data, it is currently unclear whether big data analytics' frameworks will require high-performance and large-capacity memory to cope with this change and exactly what role main memory subsystems will play; particularly in terms of energy efficiency. In this work, we investigate how the choice of DRAM (high-end vs low-end) impacts the performance of Hadoop, Spark, and MPI based big data workloads in the presence of different storage types on a local cluster. Our results show that Hadoop workloads do not require high capacity memory. However, Spark and MPI based workloads require large capacity memory. Moreover, Increasing memory bandwidth through the increasing memory frequency or the number of channels does not improve the performance of Hadoop workloads while iterative tasks in Spark and MPI benefits from high bandwidth memory. Among the configurable parameters, our results indicate that increasing the number of DRAM channels reduces DRAM power and improves the energy-efficiency across all applications.

MeNa: A Memory Navigator for Modern Hardware in a Scale-out Environment

Scale-out infrastructure such as Cloud is built upon a large network of multi-core processors. Performance, power consumption, and capital cost of such infrastructure depend on the overall system configuration including number of processing cores, core frequency, memory hierarchy and capacity, number of memory channels, and memory data rate. Among these parameters, memory subsystem is known to be one of the performance bottlenecks, contributing significantly to the overall capital and operational cost of the server. Also, given the rise of Big Data and analytics applications, this could potentially pose an even bigger challenge to the performance of cloud applications and cost of cloud infrastructure. Hence it is important to understand the role of memory subsystem in cloud infrastructure and in

particular for this emerging class of applications. In this work, through a comprehensive real-system empirical analysis of performance, we first characterize diverse types of scale-out applications across a wide range of memory configuration parameters. The characterization helps to accurately capture applications' behavior and derive a model to predict their performance. Based on the developed predictive model, we propose MeNa, which is a methodology to maximize the performance/cost ratio of scale-out applications running in cloud environment. MeNa navigates memory and processor parameters to find the system configuration for a given application and a given budget, to maximum performance. Compared to brute force method, MeNa achieves more than 90% accuracy for identifying the right configuration parameters to maximize performance/cost ratio. Moreover, we show how MeNa can be effectively leveraged for server designers to find architectural insights or subscribers to allocate just enough budget to maximize performance of their applications in the cloud.

E-Net: Energy-Aware and Neural Network-based Resource Provisioning for In-Memory Computing on Scale-Out Platform

In this work [74], we propose E-Net which leverages an artificial neural network to build a cross-platform energy-performance estimation model as well as an application's behavior predictor. Based on the developed predictive model and energy-performance estimator, E-Net uses an optimization engine to distinguish close-to-optimal configuration in order to minimize the Energy Delay Product (EDP) metric, which indicates the trade-off between energy and performance. Compared to Oracle cluster management system, proposed E-Net achieves 93% accuracy to predict the future application's behavior. E-Net shows promising energy-efficiency results, improving the EDP by 40% compared to the default scheduler in Spark and Flink and 16% compared to the state of the art technique.

ProMLB: Adaptive Performance Modeling of Data-Intensive Workloads for Resource Provisioning in Virtualized Environment

Despite an increasing interest in resource provisioning, little works have been done in developing accurate and practical models to proactively predict the performance of data-intensive applications corresponding to the server configuration and provision a cost-optimal configuration online. In this work, we introduce ProMLB: a proactive machine-learning-based methodology for resource provisioning.

ProMLB builds a set of cross-platform performance models for each application. Based on the developed predictive model, ProMLB uses an optimization technique to distinguish close-to-optimal configuration in order to minimize the product of execution time and cost. Compared to the oracle scheduler, ProMLB achieves 91% accuracy in terms of application-resource matching. On average, ProMLB improves the performance and resource utilization by 42.6% and 41.1%, respectively, compared to baseline scheduler. Moreover, ProMLB improves the performance per cost by 2.5× on average.

Adversarial Railroading of Resource Sharing-based Attacks on the Cloud

We propose Cloak & Co-locate – a novel approach to improve the effectiveness of distributed attacks on cloud infrastructure. For this purpose, by reverse-engineering the resource provisioning system and employing the adversarial machine learning attack, we co-locate adversary VM with the victim and evade detection, as well as migration caused by the scheduler. We proposed to use a fake trace generator (FTG) and wrap it around the adversary kernel (Cloak). The fake trace generator can be spawned as a separate thread, generating a pattern close to the victim VM's pattern, fooling the scheduler to co-locate it with the victim VM. After co-location, FTG continuously crafts new behavior to disguise itself and fool RPS for remaining co-located on the same host as the victim. This research motivates real-world

public cloud providers to introduce stricter isolation solutions in their platforms and systems architects to develop robust RPSs that provide security and performance predictability at high utilization.

1.4 Organization

This dissertation incorporates our previously published work [72, 69, 74, 67] and is organized as follows.

Chapter 2 surveys existing characterization of data-intensive applications and resource management mechanisms.

Chapter 3 evaluates the impact of the memory parameters on the performance and energy efficiency of big data analytics frameworks.

Chapter 4 introduces MeNa. MeNa is a three-stage methodology that navigates memory and CPU parameters to find the best performance/cost configuration for a given budget set by the user.

Chapter 5 describes E-Net, a configuration tuning methodology that automatically adjusts the hardware configuration assigned to a Virtual Machine (VM) in a proactive manner in order to dynamically optimize the energy efficiency of a given IMC program running on a given heterogeneous cluster of servers.

Chapter 6 presents ProMLB, a proactive online resource provisioning methodology to address the challenge of resource allocation for data-intensive workloads in scale-out platforms.

Chapter 7 covers the security vulnerability of resource provisioning systems and shows that RPSs can be exploited to solve the challenge of co-location in resource sharing-based attacks

in the cloud.

Chapter 8 concludes with future directions for research.

Chapter 2

State of the Arts on ML-Assisted Resource Provisioning Systems

This chapter presents the state-of-the-art resource provisioning systems. But before that, we summarize the relevant literature on the characterization of data-intensive applications and memory subsystem.

2.1 Characterization

Memory

A recent work on big data [27] profiles the memory access patterns of Hadoop and noSQL workloads by collecting memory DIMM traces using special hardware. This study does not examine the effects of memory frequency and number of channels on the performance of the system. A more recent work [21] provides a performance model that considers the impact of memory bandwidth and latency for big data, high-performance, and enterprise workloads. The work in [4] shows how Hadoop workload demands different hardware resources. This work also studies the memory capacity as a parameter that impacts the performance. However, as we showed in this work, their finding is in contrast with ours. In [141] the authors

evaluate contemporary multi-channel DDR SDRAM and Rambus DRAM systems in SMT architectures. The work in [9] mainly focuses on page table and virtual memory optimization of big data and [53] presents the characterization of cache hierarchy for a Hadoop cluster. These works do not analyze the DRAM memory subsystem. In addition, several studies have focused on memory system characterization of various non-big data workloads such as SPEC CPU or parallel benchmark suites [8, 133, 103]. Moreover, [68] studied the impact of memory parameters on the power and energy efficiency of big data frameworks but did not study the effect of input size and processor configuration on memory behavior. Another recent work studied the effect of memory bandwidth on the performance of MapReduce frameworks and presented a memory navigator for modern hardware [69]. Few works [70, 131] studied the impact of fault tolerant techniques on the performance and memory usage of embedded system.

Big Data

A recent work on big data benchmarking [124] analyzes the redundancy among different big data benchmarks such as ICTBench, HiBench and traditional CPU workloads and introduces a new big data benchmark suite for spatio-temporal data. The work in [89] selects four big data workloads from the BigDataBench [119] to study I/O characteristics, such as disk read/write bandwidth, I/O devices utilization, average waiting time of I/O requests, and average size of I/O requests. Another work [63] studies the performance characterization of Hadoop and DataMPI, using Amdahl's second law. This study shows that a DataMPI is more balanced than a Hadoop system. In a more recent work [47] the authors analyze three SPEC CPU2006 benchmarks (libquantum, h264ref, and hmmer) to determine their potential as big data computation workloads. The work in [11] examines the performance characteristics of three high-performance graph analytics. One of their findings is that graph workloads fail to fully utilize the platform's memory bandwidth. In a recent work [54], Principle Component Analysis is used to detect the most important characteristics of big data workloads from

BigDataBench. To understand Spark's architectural and micro-architectural behaviors, a recent work evaluates the benchmark on a 17-node Xeon cluster [55]. Their results show that Spark workloads have different behavior than Hadoop and HPC benchmarks. Again, this study does not consider the effect of memory subsystems on big data. The work in [50] performs performance analysis and characterizations for Hadoop K-means iterations. This study has also proposed a performance prediction model in order to estimates performance of Hadoop K-means iterations, without considering the memory requirements. The results of the latest works on memory characterization of Hadoop applications also are in-line with our findings [75, 71]. Moreover, there are studies on hardware acceleration of Hadoop applications that do not analyze the impact of memory and storage on the performance [87, 86]. Makrani et al. proposed compressive sensing based accelerator for multimedia big data application to reduce the I/O bottleneck for getting performance gain from high-end memory [73].

2.2 Resource Provisioning Systems

Table 2.1 summarizes the recent works and differentiates them from each others. In the system column, after the name of each system, we have provided the name and the conference in which the research has been published. Moreover, in this table proactive means to act before a significant change occurs in the behavior of application and influences the performance of the system.

One of the most popular RPS is Quasar [26] that leverages machine learning and collaborative filtering to quickly determine which applications can be co-scheduled on the same machine without destructive interference. CherryPick [2] is another successful system that leverages Bayesian Optimization and Regression technique to build performance models for various applications to distinguish the close-to-the-best configuration. Ernest [112] uses common machine learning kernels and statistical techniques for selecting the optimal configuration on the cloud. PARIS [127] is another ML-assisted system that uses Random Forest

Table 2.1: Comparison of state of the arts

System	Target	Complexity	Accuracy	Proactive	Dynamic	Domain	Cost aware
ProMLB (TOMPECS'21)	Performance/cost, Fairness	High	High	Yes	Yes	Big Data	Yes
BoPF (SIGMETRICS'19)	Fairness	Medium	High	No	No	Big Data	No
DAC (ASPLOS'18)	Performance	High	High	No	No	In-memory	No
PARIS (SoCC'17)	Performance	Medium	Medium	No	Yes	Broad	Yes
CherryPick (NSDI'17)	Performance	Low	Low	No	No	Big Data	No
MeNa (IISWC'17)	Performance/cost	Low	Low	No	No	Broad	Yes
HCloud (ASPLOS'16)	Cost	Medium	Medium	No	Yes	Scale-out	Yes
Ernest (NSDI'16)	Performance	Medium	High	No	No	Big Data	No
Heracles (ISCA'15)	Performance	Low	Medium	No	Yes	Latency-critical	No
Quasar (ASPLOS'14)	Performance	Medium	Medium	No	Yes	Scale-out	No
REF (ASPLOS'14)	Fairness	low	Low	No	Yes	Broad	No
Paragorn (ASPLOS'13)	Performance	Medium	Low	No	Yes	Scale-out	No

for predicting performance from the application's micro-architectural behavior to find the best VM type configuration.

There are several works that have focused on other aspects of resource provisioning such as energy efficient resource provisioning for cloud. Zhang et al. [135] provided a control-theoretic solution to the dynamic capacity provisioning problem that minimizes the total energy-cost while meeting the performance objective in terms of task scheduling delay. Guevara et al. [33] studied how heterogeneous platforms bring energy-efficiency for cloud applications. Guenter et al. [32] proposed an automated server provisioning system that aims to meet workload demand while minimizing energy consumption in data centers. Altomare et al. [3] developed a system for energy-aware allocation of virtual machines on Cloud physical nodes.

Paragon, ANN-Dynamic, and HCloud [25, 57, 23] were proposed to address QoS-aware, performance-aware, and cost-aware scheduling and resource allocation. The works in REF [132] and BoPF [61] are resource provision methods to schedule a fair set of resources for each user at a computer architecture to cloud level by presenting fair resource allocation mechanisms that customized preferences to determine each user's fair share of the hardware. However those methods are not proactive. Kulkarni et al. [58], and Delimitrou et al. [22] are other works that target heterogeneity and security of the cloud respectively. Kousiouris et al. [57] proposed to use a two-layer service in cloud to translate high level application parameters (workload and QoS based on Service Level Agreement) to resource level attributes. Their

work did not consider any performance model to select the optimum configuration. Also, they have not considered the cost efficiency.

There are other systems that adaptively allocate resources based on feedback. Rightscale [94] creates additional VM instances when the load of an application crosses a threshold for EC2. YARN [111] decides resource needs based on requests from the application. Other systems have explicit models to inform the control system, e.g., work of Bodik et al. [16]. Wrangler [126] identifies overloaded nodes in map-reduce clusters and delays scheduling jobs on them. Interference is creating challenge in accurate performance estimation. In two recent works, Maji et al. [66], and Romero et al. [96] explore placing applications on particular resources to reduce interference, by co-scheduling applications with disjoint resource requirements. However, users requesting VM types in cloud services like Amazon EC2 cannot usually control what applications are co-scheduled. None of these studies have focused on the influence of system parameters such as of memory or storage on the performance and cost in the cloud.

Moreover, Jackson et al. [51], and Barker et al. [7] analyzed high-performance computing (HPC) applications, latency-sensitive applications, scientific applications, and micro-benchmark applications on the cloud. Kanev et al. [56] analyzed cloud-scale workloads to provide infrastructure-level insights for cloud providers. Our approach is to alleviates the need for significant knowledge about the application. Using application's architectural signature collected by running a task from a workload for a short perio of time on a reference VMs, we predict the performance-energy trade-off tailored to that workload across various options in the scale-out environments.

Chapter 3

Performance Analysis of Data-Intensive Applications

The emergence of data analytics frameworks requires computational resources and memory subsystems that can naturally scale to manage massive amounts of diverse data. It is currently unclear whether big data frameworks such as Hadoop, Spark, and MPI will require high bandwidth and large capacity memory to cope with this change. The primary purpose of this chapter is to answer this question through empirical analysis of different memory configurations available for commodity server and to assess the impact of these configurations on the performance Hadoop and Spark frameworks, and MPI based applications. Our results show that neither DRAM capacity, frequency, nor the number of channels play a critical role on the performance of all studied Hadoop as well as most studied Spark applications. However, our results reveal that iterative tasks (e.g. machine learning) in Spark and MPI are benefiting from a high bandwidth and large capacity memory.

3.1 Introduction

The information age brought along an explosion of big data from multiple sources in every aspect of our lives [142]. Big data is an enabler of future strategies and immediate change through the power of predictive analytics and advanced data science. Properly harnessing data can help to achieve better, fact-based decision-making and improve the overall customer experience. By using new big data technologies, companies can answer questions in seconds rather than days, and in days rather than months. This acceleration allows businesses to enable the type of quick reactions to key business questions and challenges that can build competitive advantage and improve performance, and provide answers for complex problems or questions that have resisted analysis.

Big data analytics applications heavily rely on machine learning and data mining algorithms, and are running complex software stack with significant interaction with I/O and OS, and exhibit high computational intensity and I/O intensity [13]. In addition, unlike conventional CPU applications, big data applications combine a high data rate requirement with high computational power requirement, in particular for real-time and near-time performance constraints.

Big data frameworks such as Hadoop, Spark, and MPI are three popular platform that enables big data analytics. Hadoop has been developed to use a cluster of commodity server to process large datasets. However, Spark is developed to overcome the limitation of Hadoop on efficiently utilizing main memory. MPI, a de facto industry standard for parallel programming on distributed memory systems, is also another platform used for data analytics [119].

In the era of big data, it is important to evaluate the effect of main memory parameters on the performance of data-intensive applications in the presence of different storage types. While there is literature on understanding the behavior of big data applications by characterizing them, most of prior works have focused on the CPU parameters such as core counts, core frequency, cache parameters, and network configuration or I/O implication with the

assumption of the demand for using the fastest and largest main memory in the commodity hardware [27, 4, 53, 89, 63, 50, 76]. .

In this chapter, we evaluate the impact of the memory parameters on the performance and energy efficiency of big data analytics frameworks. To perform the memory subsystem analysis, we have investigated three configurable memory parameters including memory capacity, memory frequency, and number of memory channels, to determine how these parameters affect the performance and power consumption of big data applications. Additionally, we study the impact of storage on the memory behavior of big data applications. This analysis helps in making architectural decision such as what memory architecture to use to build a server for big data applications.

Our evaluation reveals that Hadoop applications do not require a high bandwidth-capacity memory subsystem to enhance the performance. Improving memory subsystem parameters beyond 1866 MHz Frequency and a single channel does not enhance Hadoop performance noticeably. Moreover, Hadoop framework does not require large capacity memory, since it stores all intermediates data on the storage rather than in the main memory. On the other hand, Spark and MPI applications benefit from higher memory frequency and number of channels if the application is iterative such as machine learning algorithms. However, increasing the number of memory channels beyond two channels does not enhance the performance of those applications. This is an indication for lack of efficient memory allocation and management in both hardware (memory controller) as well as software stack.

Furthermore, our results show that the memory usage of Spark framework is predictable that helps to not over-provision the memory capacity for Spark based big data applications. On the other hand, MPI framework shows that its memory capacity requirement varies significantly across studied applications. This therefore indicates that applications implemented with MPI are requiring a large capacity memory to prevent from becoming a performance bottleneck. To understand whether our observations on memory subsystem behavior remains valid for future architectures with higher number of cores, larger cache capacity, and higher operating frequency, we performed further micro-architectural study

to understand the impact of these parameters on memory behavior. Our results suggest to use a low frequency DRAM memory with high number of channels which reduces the power consumption of DRAM by 57% without any performance degradation in order to improve the energy efficiency of big data clusters.

The findings of this study are important as they help server designers to avoid over provisioning the memory subsystem for many of data analytics applications. Moreover, we found that the current storage systems are the main bottleneck for the studied applications hence any further improvement of memory and CPU architecture without addressing the storage problem is a waste of money and energy.

The remainder of this chapter is organized as follows: Section 3.2 provides technical overview of the investigated workloads and the experimental setup. Results are presented in Section 3.3. Finally, Section 3.4 concludes the chapter.

3.2 Experimental Setup

In this section, we present our experimental system configurations and its setup. We first introduce the studied frameworks and workloads. We then describe our hardware platform. Finally, we present experimental methodology and the tuning of HDFS (Hadoop Distributed File System) block size in order to optimize the platform for Hadoop and Spark frameworks.

Frameworks

Hadoop: One of the most popular framework for big data is MapReduce introduced by Google. Apache Hadoop is a java-based open source implementation of the MapReduce programming model, which is pivotal in big data computing [107]. Hadoop has been utilized in various areas, such as machine learning, search engines, log analysis, and e-commerce. The success of Hadoop is due to its scalability, fault tolerance, and simplicity of programming. Hadoop is composed of two layers. The first layer is a data storage called HDFS and the

second layer is a data processor called Hadoop MapReduce framework. HDFS is a block based file system. Hadoop MapReduce cannot keep reused data and state information during execution [46]. Hence, it has to iteratively read the same data in each iteration, which results in significant disk accesses and unnecessary overhead.

Spark: Spark is another MapReduce-like cluster computing framework designed to overcome Hadoop's shortage in utilizing main memory. Spark uses HDFS as data storage system. In addition, Spark uses a new data structure called Resilient Distribute Dataset (RDD). The main responsibility of RDD is to cache data, which avoids data reloading from the disk. RDD allows users to cache the high value data in memory, and controls the persistence of data. It is suitable for applications with iterative algorithms that can achieve tremendous speed up. Moreover, Spark supports a Directed Acyclic Graph (DAG) schedule which avoids materializing the intermediate values by pipeline operations to decrease I/O accesses. While Hadoop uses a heartbeat scheduler to communicate scheduling decisions which impose 5 to 10 second delay, Spark task scheduling is low latency through an event-driven architecture.

MPI: A peer-to-Peer network is a decentralized and distributed network where thousands of machines connected in the network consume, as well as, serve resources. Nodes use Message Passing Interface (MPI) to communicate and exchange data between themselves. One of the features of MPI is that a process does not need to read same data over and over because it can live as long as the system runs. However, MPI has a major drawback of lacking fault tolerance. Each node in this network is a single point of failure that can cause the whole system to shut down. Hence, users who prefer a robust and fault tolerant framework exploit other big data framework such as Hadoop. In response to this issue, there are plans to include fault-tolerance inside the MPI model in its next major release. In this chapter, our focus is not on MPI applications but on data analytics workloads which use MPI for parallel implementation. The nature of most of big data applications is simple but the main challenge is to process big amount of data which is out of the capability of a single server to process. Therefore, findings of this chapter regarding MPI are only valid for studied applications. It is important to note that, there are complex MPI based applications

(outside of the scope of this study) that could have completely different behavior than what we have observed in our experiments.

In our study, we used Hadoop MapReduce version 2.7.1, Spark version 2.1.0 in conjunction with Scala 2.11, and MPICH2 version 3.2 installed on Linux Ubuntu 16.04 LTS. Our JVM version is 1.8.

Table 3.1: Studied workloads

<table>
<tr><th>Workload</th><th>Domain</th><th>Input type</th><th>Input size (huge)</th><th>Framework</th><th>Suite</th></tr>
<tr><td>Wordcount</td><td>micro-kernel</td><td>text</td><td>1.1 TB</td><td rowspan="3">Hadoop, Spark, MPI</td><td rowspan="6">BigData Bench</td></tr>
<tr><td>Sort</td><td>micro-kernel</td><td>data</td><td>178.8 GB</td></tr>
<tr><td>Grep</td><td>micro-kernel</td><td>text</td><td>1.1 TB</td></tr>
<tr><td>Terasort</td><td>micro-kernel</td><td>data</td><td>834 GB</td><td>Hadoop, Spark</td></tr>
<tr><td>Naive Bayes</td><td>E-commerce</td><td>Data</td><td>306 GB</td><td>Hadoop, Spark, MPI</td></tr>
<tr><td>Page Rank</td><td>E-commerce</td><td>Data</td><td>306 GB</td><td>Hadoop, Spark</td></tr>
<tr><td>Bayes</td><td>E-commerce</td><td>Data</td><td>306 GB</td><td>Hadoop, Spark</td><td>HiBench</td></tr>
<tr><td>k-means</td><td>Machine learning</td><td>Graph</td><td>112.2 GB</td><td>Hadoop, Spark, MPI</td><td>BigDataBench</td></tr>
<tr><td>nweight</td><td>Graph analytics</td><td>Graph</td><td>176 GB</td><td>Spark</td><td rowspan="4">HiBench</td></tr>
<tr><td>Aggregation</td><td>Analytical query</td><td>Data</td><td>1.08 TB</td><td rowspan="3">Hadoop</td></tr>
<tr><td>Join</td><td>Analytical query</td><td>Data</td><td>1.08 TB</td></tr>
<tr><td>Scan</td><td>Analytical query</td><td>Data</td><td>1.08 TB</td></tr>
<tr><td>B.MPEG</td><td>Multimedia</td><td>DVD stream</td><td>437 GB</td><td rowspan="6">MPI</td><td rowspan="6">BigDataBench</td></tr>
<tr><td>DBN</td><td>Multimedia</td><td>Images</td><td>MNIST Dataset</td></tr>
<tr><td>Speech recognition</td><td>Multimedia</td><td>Audio</td><td>252 GB</td></tr>
<tr><td>Image segmentation</td><td>Multimedia</td><td>Images</td><td>162 GB</td></tr>
<tr><td>SIFT</td><td>Multimedia</td><td>Images</td><td>162 GB</td></tr>
<tr><td>Face detection</td><td>Multimedia</td><td>Images</td><td>162 GB</td></tr>
</table>

Workloads

Big data analytics applications are characterized by four critical features, referred as the four "Vs": volume, velocity, variety, and veracity. Big data is inherently large in volume. Velocity refers to how fast the data is coming in and to how fast it needs to be analyzed. Variety refers to the number and diversity of sources of data and databases, such as sensor data, social media, multimedia, text, and much more. Veracity refers to the level of trust, consistency, and completeness of data. The diversity of applications is important for characterizing big data frameworks. This diversity can enable users to optimize their programs considering the memory configuration of the framework.

Similarly, cluster designers can evaluate their candidate memory configurations by considering different classes of applications. Hence, for this study we target various domains of applications namely that of microkernels, graph analytics, machine learning, E-commerce, social networks, search engines, and multimedia. We used BigDataBench [119] and HiBench [46] for the choice of benchmarking. We selected a diverse set of applications and frameworks to be representative of data analytics domain. More details of these workloads are provided in Table 5.1. The selected workloads have different characteristics such as high level data graph and different input/output ratios. Some of them have unstructured data type and some others are graph based. Also these workloads are popular in academia and are widely used in various studies.

Hardware platform

We carefully selected our experimental platform to investigate the micro-architectural effect on the performance of data analytics frameworks to understand whether our observations on memory subsystem behavior remains valid for future architectures with enhanced microarchitecture parameters or not. This includes analyzing the results when increasing the core count and processor operating frequency. This is important, as the results will shed light on whether in future architectures larger number of cores, higher cache capacity and higher operating frequency change memory behavior of big data applications or not. Using the data collected from our experimental test setup, we will drive architectural conclusion on how these microarchitecture parameters are changing DRAM memory behavior and therefore impacting performance and energy-efficiency of data-intensive applications.

For running the workloads and monitoring statistics, we used a six-node standalone cluster with detailed characteristics presented in Table 7.1. To have a comprehensive experiment we used different SDRAM memory modules. All modules are provided from the same vendor. We used single socket servers in this study, in order to hide the NUMA effect (to understand DRAM-only impact). While network overhead in general is influencing the performance of

studied applications and therefore the characterization results, for big data applications, as shown in a recent work [88], a modern high speed network introduces only a small 2% performance benefit. We therefore used a high speed 1 Gbit/s network to avoid making it a performance bottleneck for this study. Our NICs have two ports and we used one of them per node for this study.

Table 3.2: Hardware Platform

Hardware type	Parameter	Value
CPU	Model	Intel Xeon E5-2683 V4
	# Core	16 (32 thread)
	Base Frequency	2.1 GHz
	Turbo Frequency	3.0 GHz
	TDP	120
	L3 Cache	40 MB
	Memory Type Support	DDR4 1866/2133/2400
	Maximum Memory Bandwidth	76.8 GB/S
	Max Memory Channels supported	4
Disk (SSD PCIE)	Model	Samsung 960 PRO M.2
	Capacity	512 GB
	Speed	Max 3.5 GB/S
Disk (SSD SATA)	Model	HyperX FURY
	Capacity	480 GB
	Speed	500 MB/S
Disk (HDD)	Model	Seagate
	Capacity	500 GB
	Speed	7200 RPM
Network Interface card	Model	ST1000SPEXD4
	Speed	1000 Mbps

Methodology

The experimental methodology of this chapter is focused on understanding how data analytics frameworks are utilizing main memory.

Data collection: We used Intel Performance Counter Monitor tool (PCM) [43] to understand hardware (memory and processor) behavior. The performance counter data are

collected for the entire run of each application, those counters were used to get the amount of Bytes read or written by memory controller to calculate the memory bandwidth. We collect OS-level performance information with DSTAT tool-a profiling tool for Linux based systems by specifying the event under study. Some of the metrics that we used for this study are memory footprint, L2, and Last Level Cache (LLC) hits ratio, instruction per cycle (IPC), core C0 state residency, and power consumption. For power measurement, we used PCM-power utility, which provides the detailed power consumption of each socket and DRAM. We did not use WattsUp power meter because it does not have the breakdown of power and also it collects power consumption of several parts of the system which are not related to this study. Throughout this chapter we will present the results based on high speed SSD disk. The default values for experiments are as follow: DRAM capacity = 32 GB, number of memory channels = 4, memory frequency = 2400 MHz, core count per CPU = 16, and CPU frequency = 2.6 GHz.

Parameter tuning: For both Hadoop and Spark frameworks, it is important to set the number of mapper and reducer slots appropriately to maximize the performance. Based on the result of [29], the maximum number of mappers running concurrently on the system to maximize performance should be equal to the total number of available CPU cores in the system. Therefore, for each experiment, we set the number of mappers equal to the total number of cores. We also follow same approach for the number of parallel tasks in MPI. Adjusting default memory parameters of Hadoop and Spark also is important. Hence, we tuned Hadoop and Spark memory related configuration parameters. Followings are two most important memory related parameters that we tuned for all experiments:

mapreduce.map.memory.mb: is the upper memory limit that Hadoop allows to be allocated to a mapper, in megabytes. *spark.executor.memory*: Amount of memory to use per executor process in Spark (e.g. 2 GB, 8 GB).

We set those values according to the following (we reserved 20% of DRAM capacity for

OS):

$$\begin{aligned} &mapreduce.map.memory.mb = \\ &(\text{DRAM capacity} \times 0.8)/ \\ &\text{Number of concurrent mappers per node} \end{aligned} \tag{3.1}$$

$$\begin{aligned} &spark.executor.memory = \\ &((\text{DRAM capacity} - spark.driver.memory) \times 0.8)/ \\ &\text{Number of executor per node} \end{aligned} \tag{3.2}$$

A recent work has shown that among other tuning parameters in a MapReduce framework, HDFS block size is also influential on the performance [14]. HDFS Block size has a direct relation to the number of parallel tasks (in Spark and Hadoop), as shown in EQ. (3.3).

$$\text{Number of Tasks} = \text{Input Size}/\text{Block Size} \tag{3.3}$$

In the above equation, the input size is the size of data that is distributed among nodes. The block size is the amount of data that is transferred among nodes. Hence, block size has impact on the network traffic and its usage. Therefore, we first evaluate how changing this parameter affects the performance of the system. We studied a broad range of HDFS block sizes varying from 32 MB to 1GB when the main memory capacity is 64 GB per node and it has the highest frequency and number of channels. Table 7.2 demonstrates the best HDFS configuration for maximizing the performance in both Hadoop and Spark frameworks based on the ratio of Input data size to the total number of available processing cores, and the application class. The rest of the experiments presented in this chapter are based on Table 7.2 configuration. We will present the classification of applications into CPU-intensive, I/O-intensive, and memory-intensive tasks in next section. Our tuning methodology helps to put high pressure on memory subsystem.

Table 3.3: HDFS block size tuning

Application class	*Input size/(# nodes × #cores per node)*			
	<64 MB	**<512 MB**	**<4 GB**	**> 4 GB**
CPU intensive	32 MB	64 MB	128 MB	256 MB
I/O intensive	64 MB	256 MB	512 MB	1 GB
Iterative tasks	64 MB	128 MB	256 MB	512 MB

3.3 Results

Our experimental results are presented in this section. First, we present the memory analysis of the studied workloads. We present how performance of studied workloads is sensitive to memory capacity, frequency and number of channels. Then, we provide results of architectural implication of processor parameters on data analytics frameworks and memory requirements. We also discuss the impact of storage system, and size of input data on memory subsystem. In addition, we present the power analysis results. This is to help finding out which memory configuration is a better choice for energy-efficient big data processing.

Memory Analysis

In this section, we present a comprehensive discussion on memory analysis results to help better understanding the memory requirements of big data frameworks. As the focus of this study is on the memory subsystem, each memory related experiment has been performed 5 times. We present the average results, and also the minimum and maximum values of each set of experiments as an error bar.

Memory channels implication: The off-chip peak memory bandwidth equation is shown in EQ. (3.4).

$$Bandwidth = Channels \times Frequency \times Width \tag{3.4}$$

We observe in Figure 3.1 that increasing the number of channels from 1 to 4 does not significantly reduces the execution time of Hadoop applications (on average 6%). However,

the execution time of K-means and Nweight in Spark framework, and Image segmentation with MPI implementation reduces more than 30%. It is noteworthy that aforementioned workloads are iterative tasks. Figure 3.2 provides more insights to explain this exceptional behavior. This figure demonstrates the memory bandwidth usage of each workload. K-means, Image Segmentation, and Nweight memory bandwidth usages are shown to be substantially higher than other workloads. One reason is those workloads are iterative and the second reason is the low cache hit rate of these application that cause excessive access to main memory (we will present that cache hit rate later in this chapter). Therefore, providing more bandwidth improves their performance. By increasing the number of channels from 1 to 4, the performance gain is found to be 38%.

Memory frequency implication: As results in Figure 3.3 shows, similarly we don't observe significant reduction of execution time by increasing memory frequency (from 1866 MHz to 2400 MHz) for most of Hadoop applications. This finding may mislead to use the lowest memory frequency for Hadoop applications. Based on EQ. (3.5), read latency of DRAM depends on the memory frequency.

$$Read_latency = 2 \times (CL/Frequency) \tag{3.5}$$

However, for DDRx (e.g. DDR3), this latency is set fixed by the manufacturer with controlling CAS latency (CL). This means two memory modules with different frequency (1333 MHz and 1866 MHz) and different CAS Latency (9 and 13) can have the same read latency of 13.5 ns, but provide different bandwidth per channel (10.66 GB/s and 14.93 GB/s). Hence, as along as reduction of frequency does not change the read latency, it is recommended to reduce DRAM frequency for Hadoop applications. In the other hand, we observe that iterative tasks in Spark and MPI require high frequency memory and their execution time reduces by high bandwidth memory.

DRAM capacity implication: To investigate the impact of memory capacity on the performance of big data applications, we run all workloads with 7 different memory capacities per node. During our experiments, Spark workloads encountered an error when running

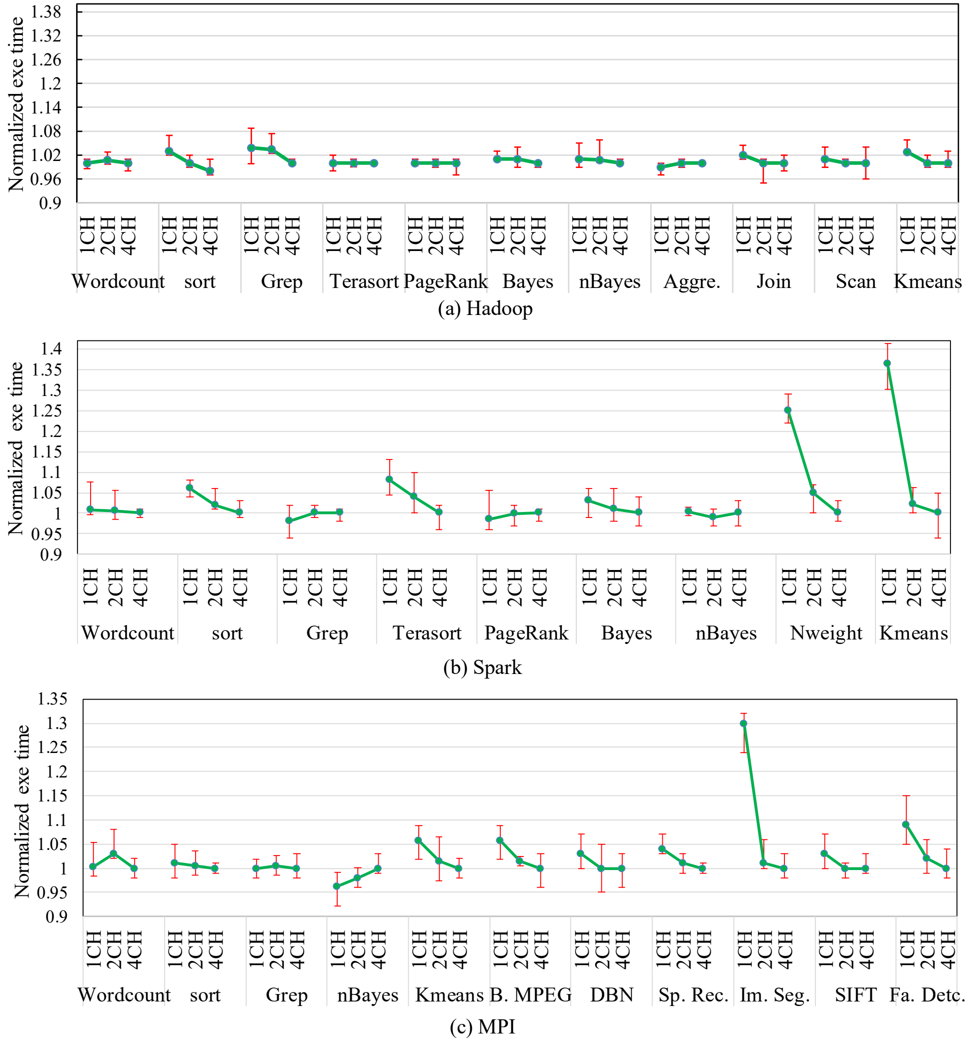

Figure 3.1: Effect of memory channel on the execution time (Normalized to 4CH)

on a 4GB memory capacity per node due to lack of memory space for the Java heap. Hence, the experiment of Spark workloads is performed with at least 8 GB of memory. An interesting observation is that a large memory capacity has not impact on the performance of studied Hadoop workloads. Hadoop applications do not require high capacity memory because Hadoop stores all intermediate values generated by map tasks on the storage. Hence, regardless of the number of map tasks or input size, the memory usage remains almost the same. In our experiments, the memory capacity usage of studied Hadoop applications never

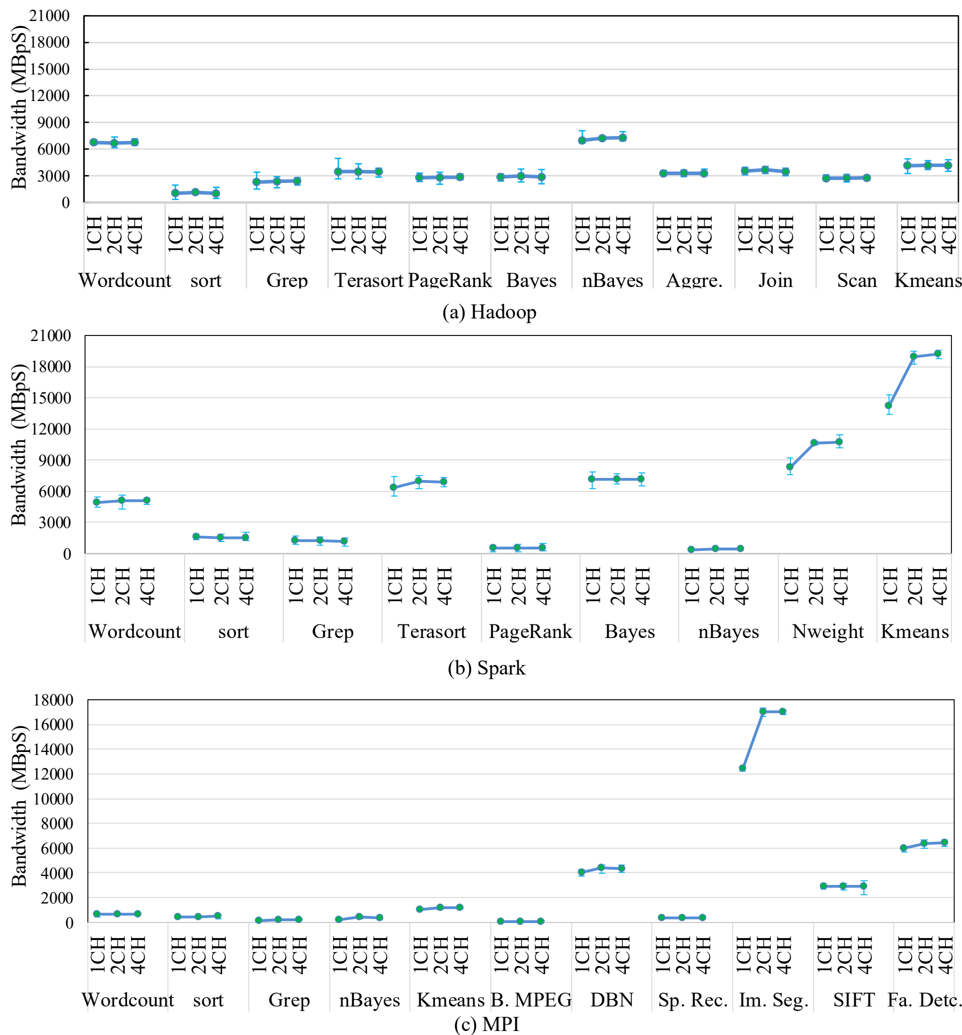

Figure 3.2: Impact of memory channel on bandwidth usage

exceeded 4 GB on each node.

However, Spark and MPI based applications show different behavior. Spark uses RDD to cache intermediate values in memory. Hence, by increasing the number of map tasks to run on a node, the memory usage increases. Therefore, by knowing the number of map tasks assigned to a nodes and the amount of intermediate values generated by each task, the maximum memory usage per node of Spark applications is predictable. To better understand the impact of memory capacity on the performance, we have provided the average normalized execution time of these three frameworks in Figure 3.4 (Normalized to 64 GB). To illustrate

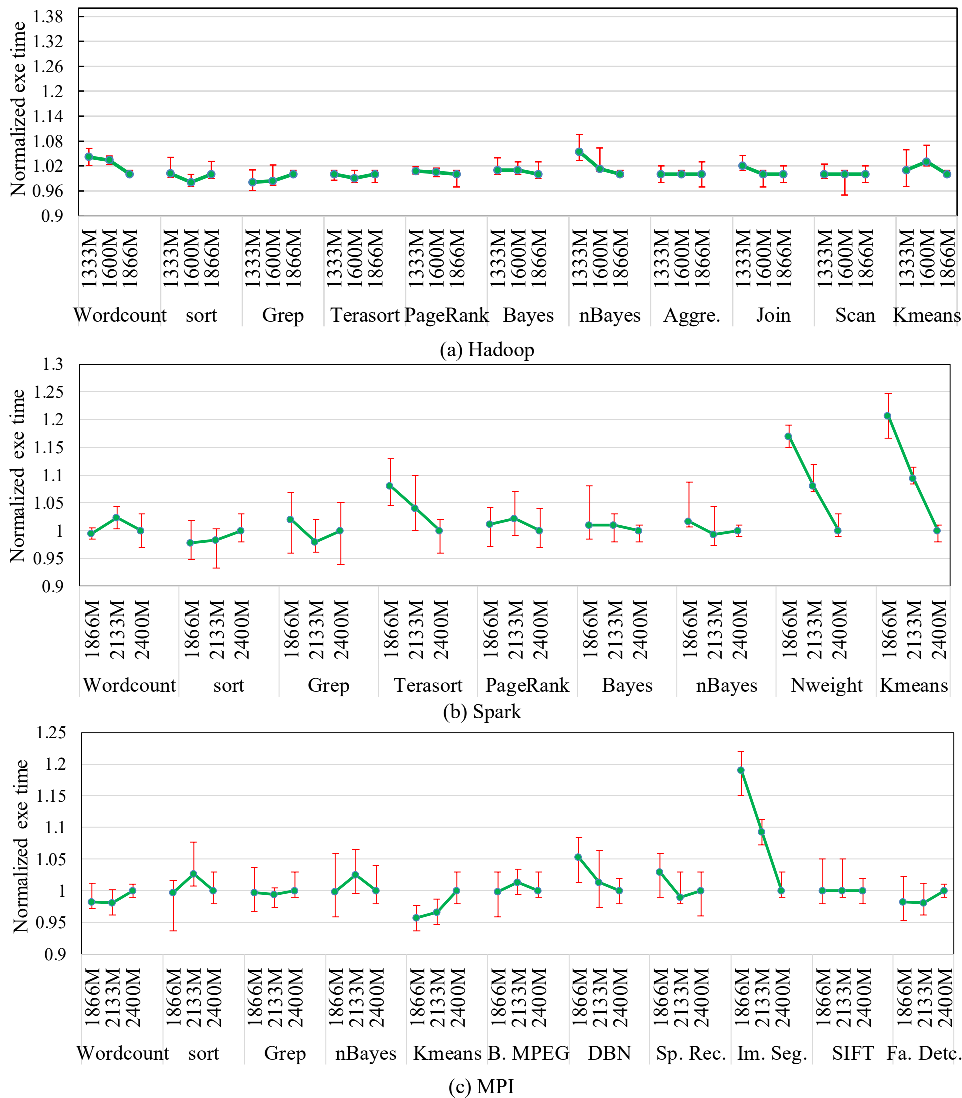

Figure 3.3: Effect of memory frequency on the execution time (Normalized to 2400MHz)

how these frameworks utilize DRAM capacity we present K-means memory usage on different frameworks in Figure 3.5.

Input size implication: Today the paradigm has been shifted and new MapReduce processing frameworks such as Hadoop and Spark are emerging. Hadoop uses disk as storage and rely on a cluster of servers to process data in a distributed manner. The ability of hadoop frameworks is that each map task processes one block of data on HDFS at a time. Hence, this relieves the pressure of large input data on the memory subsystem. Therefore, regardless

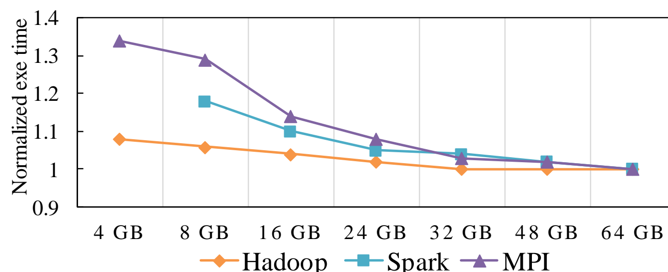

Figure 3.4: Impact of memory capacity per node on performance

of input size, the memory subsystem usage remains almost constant in this framework. In the other hand, Spark is in-memory computing framework and changing input size have a large impact on the memory capacity usage. Fo MPI based applications, the extent of impact of input size on memory capacity usage depends on the application 's implementation. Although, for most of MPI based applications, we observed that memory capacity usage increases by increasing input size.

Another parameter that can be affected by the size of input data is the memory bandwidth usage. Our results reveal that the size of input data does not noticeably change the memory behavior of big data frameworks. Because the memory bandwidth usage depends to the cache miss ratio of application (further we will discuss it in detail). Also cache behavior of application mostly depends to the application algorithm. Consequently, by increasing the size of input, the cache hit ratio remains almost the same. Therefore, while increasing the input size increases the job completion time, the DRAM bandwidth requirements of applications do not change noticeably. Figure 3.6 shows the average results of workloads from Hibench benchmark. We have performed experiments with 3 sets of input data namely medium, large, and huge (these are keywords of Hibench for input generation).

Architectural analysis

As we discussed, several parameters, such as CPU frequency, number of cores per CPU, and cache hierarchy are studied in this chapter to characterize big data frameworks. In this

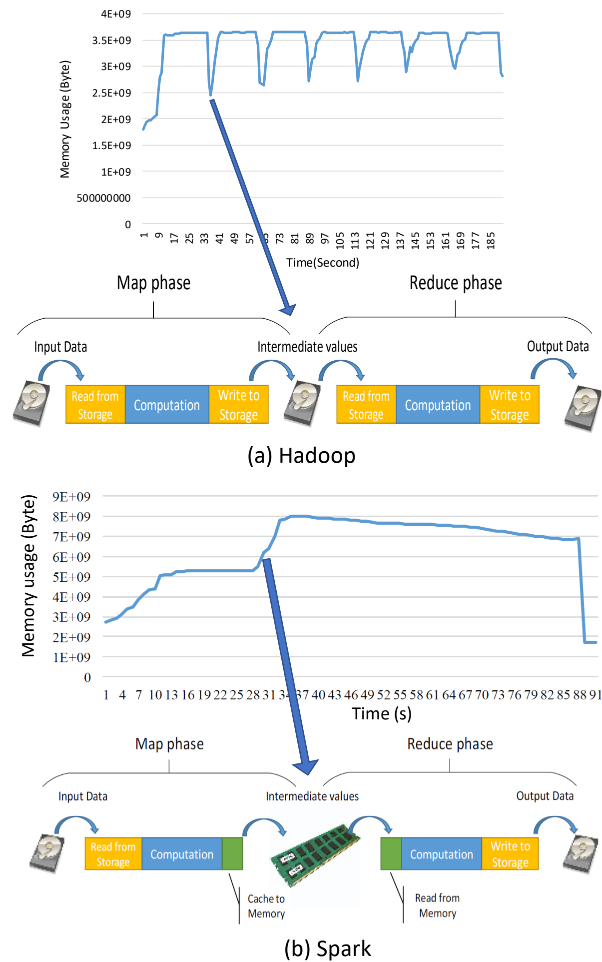

Figure 3.5: K-means memory usage on various frameworks

section, we present the classification of workloads into memory bound, compute bound, and I/O bound based on architectural behavior, which helps to accurately present the relation of performance and workload characteristics.

Classification of workloads: As the goal of this section is to study the combined impact of node architecture and data-intensive workload's characteristics, it is important to classify those workloads. To this goal, we have explored the architectural behavior of studied workloads to classify them and find more insights.

1) Core frequency implication: Figure 3.7 shows that studied workloads behave in two distinct ways. The execution time of the first group is decreased linearly by increasing the

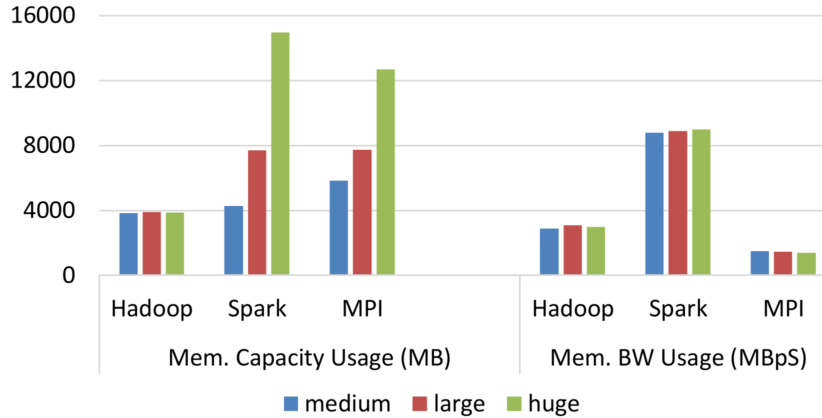

Figure 3.6: Average results of input size's effect on memory behavior

core frequency. The second group's execution time does not drop significantly by increasing the CPU frequency, particularly when changing frequency from 1.9 GHz to 2.6 GHz. These two trends indicate that studied workloads have distinct behaviors of being either CPU bound or I/O bound. This conclusion further advocated by C0 state residency of processor in Figure 3.8. This proves sort, grep, PageRank, and scan from Hadoop, wordcount, grep, PageRank, Bayes, and nBayes from Spark, and sort, BasicMPEG, and grep from MPI to be I/O bound while others to be CPU bound. This can be explained as follows: If increasing the processor's frequency reduces the active state residency (C0) of the processor, the workload is I/O bound, as when a core is waiting for I/O, the core changes its state to save power. Similarly, if active state residency does not change the workload is CPU bound.

2) Cache implication: Modern processor has a 3-level cache hierarchy. Figure 3.9 shows cache hit ratio of level 2 (L2) and last level cache (LLC). The results reveal an important characteristic of data-intensive workloads. Our experimental results show most of the studied workloads (particularly MapReduce workloads) have a much higher cache hit ratio, which helps reducing the number of accesses to the main memory. Based on simulation as well as real-system experiment results in recent works, it is reported that these applications' cache hit rate is too low (under 10%) [27, 4] for a system with 10 MB of LLC and for having LLC hit rate of 40%, the system should have around 100 MB of LLC. However, our real system

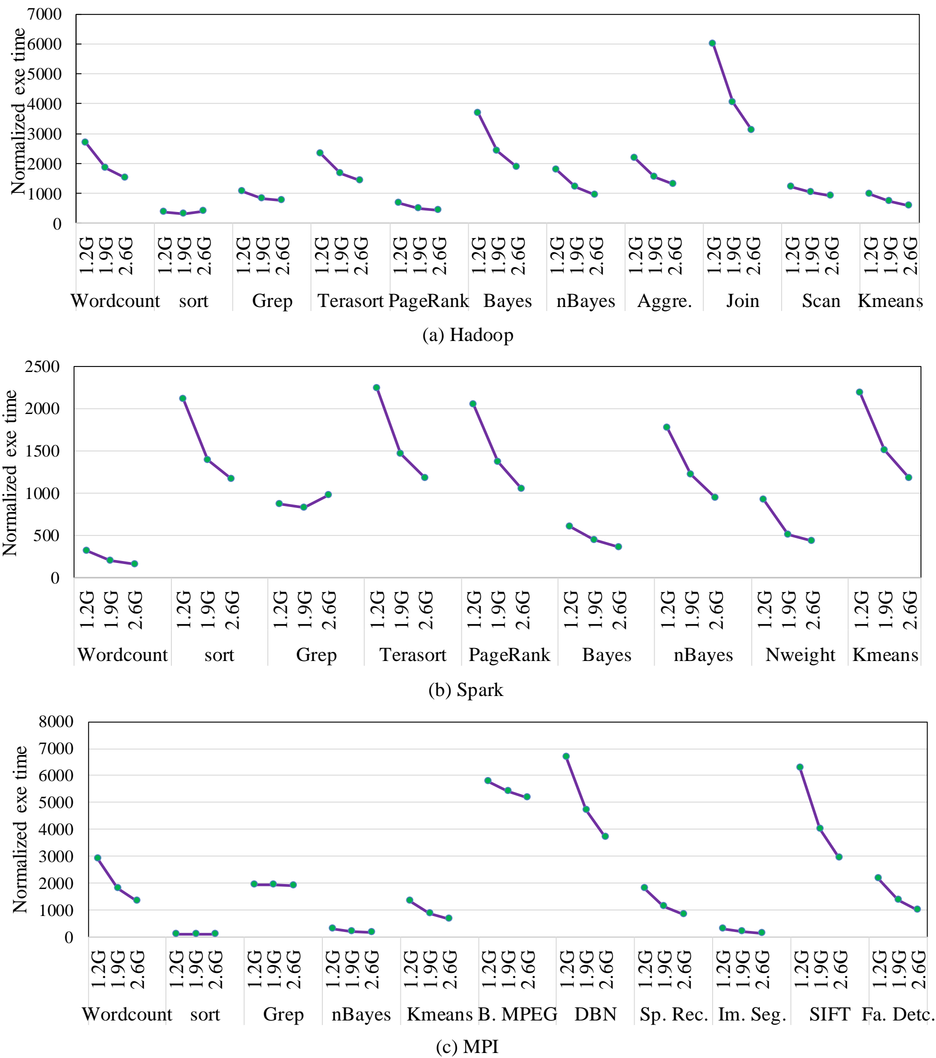

Figure 3.7: Impact of CPU frequency on the execution time

experimental results show that most of data-intensive workloads have much higher LLC hit rate (more than 50%) with only 40 MB LLC.

The reason of high cache hit ratio is that each parallel task of MapReduce framework processes data in a sequential manner. This behavior increases the cache hits; therefore it prevents excessive access to DRAM. Hence, based on the cache hit ratio of workloads and the intensity of accesses to memory, we can classify them into memory bound. If the cache hit ratio is low and the workload is an iterative task, it is classified as memory intensive. Our

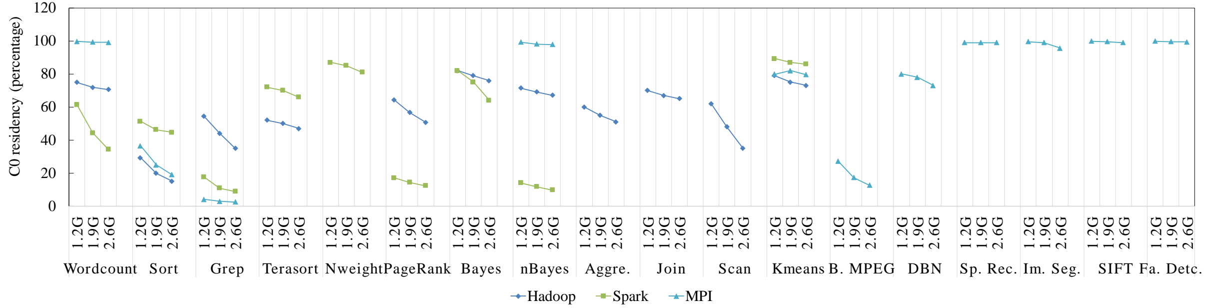

Figure 3.8: C0 residency of processor

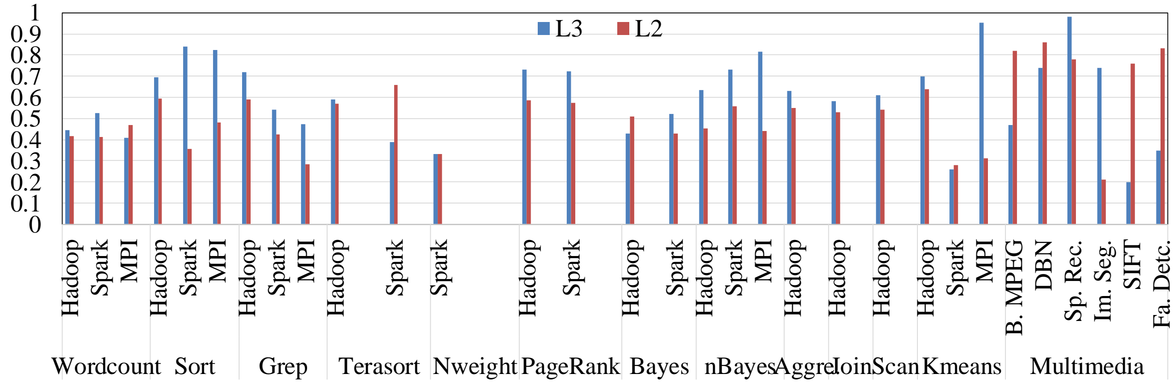

Figure 3.9: LLC and L2 hit rate

characterization showed that N-weight and Kmeans from Spark, and Image Segmentation from MPI are memory intensive. Therefore, we divided our workloads into three major groups of I/O bound, compute bound, and memory bound.

Disk implication: To show how choice of storage can change the performance while using different memory configuration, we performed several experiments using three types of storage (HDD, SSD SATA, and SSD PCIe). Figure 3.10 shows that changing the disk from HDD to SSD PCIe improves the performance of Spark, Hadoop, and MPI by 1.6x, 2.4x, and 3.3x respectively. The reason that MPI workloads take more advantage from faster disk is that these workloads are written in C++. However, Hadoop and Spark are Java based frameworks and they use HDFS as an intermediate layer to access and manage storage. Our results show a high bandwidth DRAM is not required to accelerate the performance of

MapReduce frameworks in presence of a slow HDD. However, MPI based workloads has the potential to benefit from high-end DRAM.

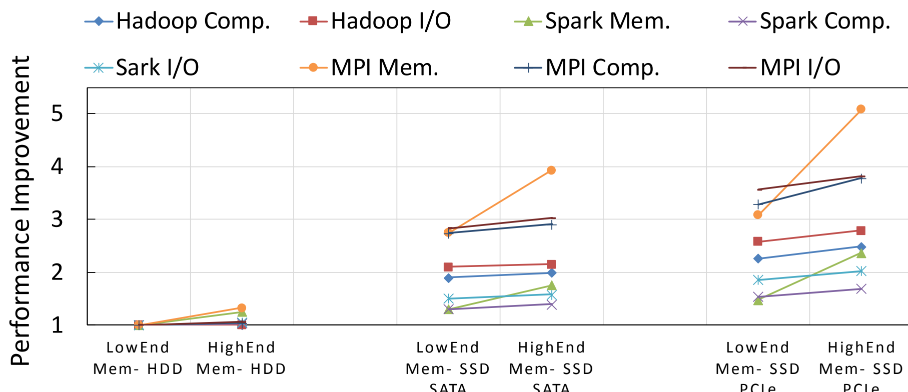

Figure 3.10: Effect of memory and storage configuration on the performance

Another point regarding the storage is to use multiple disks per node to alleviate IO bottleneck. We performed a new set of experiments with two SSD storages per node. While the performance will improve for IO intensive applications but using multiple disks per node does not guarantee the parallel access to the data blocks of HDFS to reduce the IO bottleneck. Another point is to use RAID. Since HDFS is taking care of fault-tolerance and "striped" reading, there is no need to use RAID underneath an HDFS. Using RAID will only be more expensive, offer less storage, and also be slower (depending on the concrete RAID configuration). Since the Namenode is a single-point-of-failure in HDFS, it requires a more reliable hardware setup. Therefore, the use of RAID is recommended only on the Namenodes.

It is important to note that SSD increases the read and write bandwidth of disk and substantially reduces the latency of access to disk compared to HDD. However, accessing to I/O means losing of millions of CPU cycles, which is large enough to remove any noticeable advantage of using a high-performance DRAM. On the other hand, the only way to take advantage of a SSD is to read or write a big file (hundreds of megabyte) at once but our result shows that HDFS reads the data in much smaller blocks, regardless of HDFS block size.

Figure 3.11 demonstrates the memory bandwidth utilization of each class on different storage type. Bandwidth utilization of memory bound workloads is shown to be substantially higher than other workloads and the results shows using low speed disk reduces 55% the memory bandwidth usage, and therefore, prevents to get performance benefit from high bandwidth DRAM.

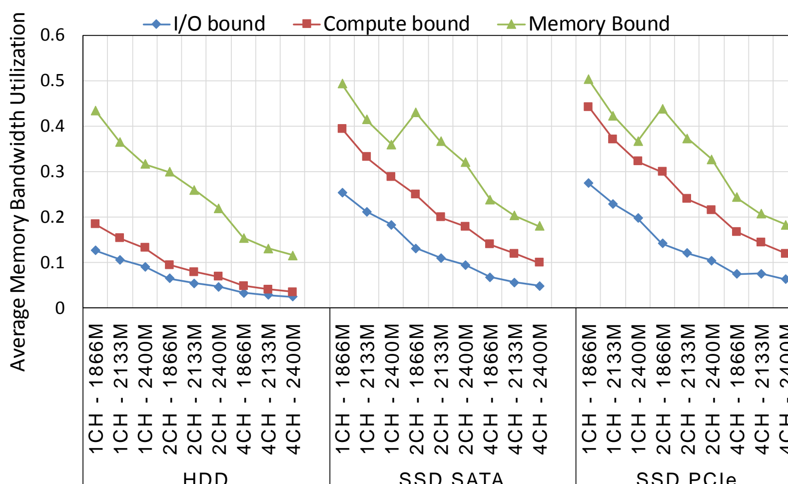

Figure 3.11: Average memory bandwidth utilization

Core count implication: In the previous section, we classified workloads into three groups. Figure 3.12 demonstrates the effect of increasing the number of cores per node on the performance of two groups of CPU intensive and IO intensive. The expectation is that performance of the system improves linearly by adding cores because big data workloads are heavily parallel. However, we observe a different trend. For CPU intensive workloads and when the core count is less than 6 cores per node, the performance improvement is close to the ideal case. The interesting trend is that increasing the number of cores per node does not improve the performance of data-intensive workloads noticeably beyond 6 cores. As the increase in the number of cores increases the number of accesses to the disk, the disk becomes the bottleneck of the system. At 8 cores, the CPU utilization is dropped to 44% for I/O intensive workloads, on average. This experiment performed when storage was SSD PCIe.

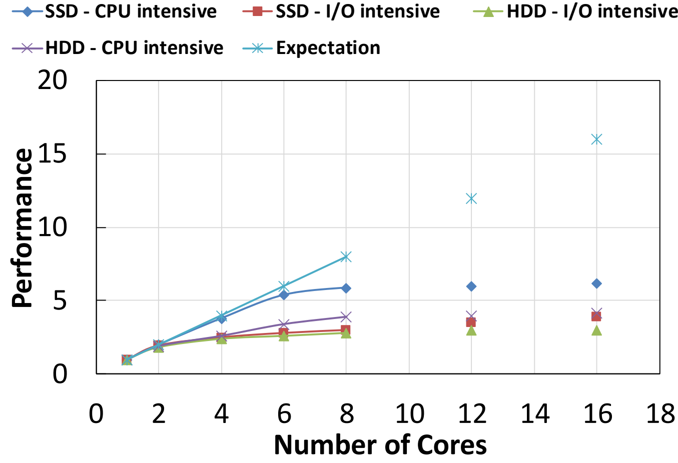

Figure 3.12: Effect of core count on the performance

Based on those observations, we develop Eq. (3.6) to find the number of cores for which further increase does not noticeably enhance the performance of system:

$$Max(cores) = ((BW \times Nd))/((Nsc \times Fr \times \lambda)) \tag{3.6}$$

We define the variables used in this equation as follow: BW is the nominal bandwidth of each disk. Nd is the number of disk installed on the server. Nsc is the number of sockets. Fr is CPU core frequency and Lambda is a constant, which we found through our real-system experiments. As the effective I/O bandwidth depends on the block size and I/O request size, we have used fio [1] to calculate Lambda for different block size requests, presented in Figure 3.13. Designers can use this equation to select an optimum configuration, such as the number of cores, core frequency, disk type, and number of disks per node. As an example, the number of cores beyond which there is no noticeable performance gain on a server with one socket, one SSD storage with nominal 400 MBpS bandwidth, 16KB IO request size, running at 2.8 GHz is 8 based on the above equation. We have validated equation 1 for different classes of workload and the result is presented in Figure 3.14.

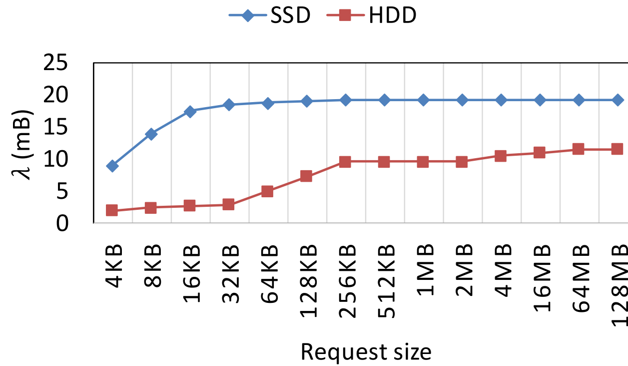

Figure 3.13: Lambda value for different request size and storage type

IO Request Size (IRS)	IRS = 16KB			32KB = IRS = 256KB			512KB = IRS = 128MB		
Workload class	Com.	Mem.	IO	Com.	Mem.	IO	Com.	Mem.	IO
Ave. error	5%	6%	15%	4%	4%	12%	4%	4%	7%

Figure 3.14: Average error of optimum core count prediction

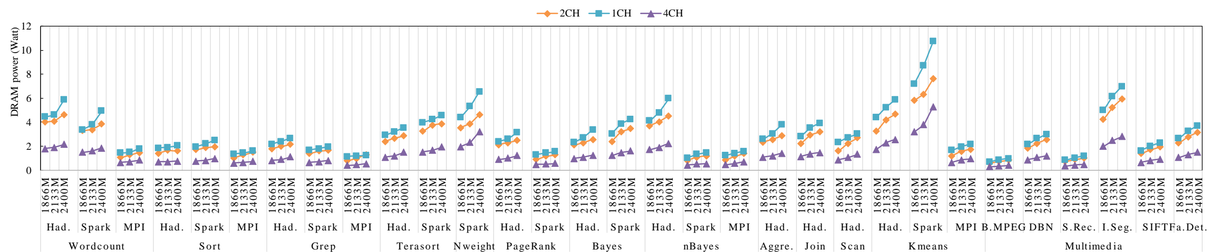

Figure 3.15: DRAM power consumption

Power Analysis

Figure 3.15 reports the DRAM power consumption. The first observation is that by increasing the frequency of DRAM by about 28% (1866 MHz to 2400 MHz), the power increases by almost 15%. Also, DRAM power increases when the core frequency raises as this increases the number of accesses to off-chip memory per unit of time. However, the DRAM power consumption is reduced when we increase the number of channels. An interesting observation is that a memory with 4 channels consumes 42% less power than a memory with 1

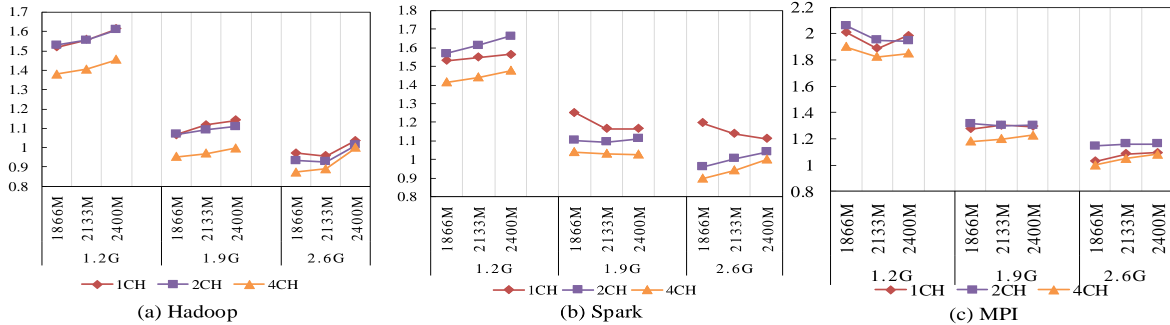

Figure 3.16: Average normalized EDP (Normalized to 4CH, 2400MHz, 2.6GHz)

channel. This is due to the fact that DRAM channels are designed to increase the efficiency of interleaving. Thus, the memory controller can manage accesses more efficiently, which in turn reduces the power consumption.

In our experiments, the number of DIMMs/Channel is one. Despite the small/no impact on performance, increasing the number of memory channels to four significantly impacts the power consumption. Reducing the power of DRAM while increasing the number of channels can be explained as follow: Consider a 32 GB and 4 channel memory (occupied 4 DIMMs with 8 GB module), the memory controller does not need to put all modules in active state unlike the single channel. In single channel memory, one module with 32 GB capacity always must be in active state regardless of memory usage pattern. However, with 4 channels, memory controller can manage each channel individually and if there is no need to access to a channel, it can go to power saving mode. This increases the options for memory controller to perform power management.

Figure 3.16 depicts the average normalized Energy Delay Product (EDP) of frameworks. EQ. (3.7) indicates how we calculated this metric.

$$EDP = (CPUenergy + DRAMenergy) \times ExecutionTime \tag{3.7}$$

The results reveal that almost always increasing the number of channels improves EDP, as memory controller power management policy can benefit from such increase and reduce the power accordingly. Increasing the number of channels does not increase the execution time

of studied applications. It also reduces DRAM power by up to 40%. Therefore, decreasing DRAM power decreases system power and consequently the energy of the system.

$$Energy = Power \times Delay \tag{3.8}$$

Hence, increasing the number of memory channels leads to an improvement in energy efficiency. The EDP's results of 1-channel and 2-channel memory are very similar. The trend in this figure shows that increasing the memory frequency increases EDP across all CPU operating frequency points. This implies that a high frequency off-chip memory is not an appropriate choice for EDP optimization for this class of big data applications. Furthermore, we observe that the EDP at 1.9 GHz CPU frequency is close to 2.6 GHZ. It shows that running the processor with highest frequency is not always necessary. A configuration with a single channel running at 2400 MHz memory frequency when CPU is running at 1.2 GHz is shown to have the worst EDP. On the other hand, a configuration with 4-channel and 1866 MHz memory frequency when CPU is running at 2.6 GHz is shown to have the best EDP for big data applications.

3.4 Conclusion

This chapter answers the important questions of whether some of important data analytics frameworks such as Hadoop, Spark and MPI require high-capacity and high-performance DRAM memory and what the role of memory for energy-efficient processing of data-intensive applications is. Characterizing memory behavior of frameworks is important as it helps guiding scheduling decision in cloud scale architectures as well as helping making decisions in designing server cluster for big data computing. While latest works have performed a limited study on memory characterization of data-intensive applications, this work performs a comprehensive analysis of memory requirements through an experimental evaluation setup. We study diverse domains of applications from microkernels, graph analytics, machine learning, E-commerce, social networks, search engines, and multimedia in Hadoop, Spark, and

MPI. This gives us several insights into understanding the memory role for these important frameworks.

The contribution of this chapter is to give an insight on the role the memory subsystem plays in the overall performance of the servers when running data analytics frameworks. Our experimental results illustrate that data-intensive workloads show three distinct behaviors (CPU-intensive, Disk-intensive, and memory-intensive). Based on the results presented in this chapter, we observed that Hadoop framework is not memory intensive. This means Hadoop does not require high frequency, and large number of channels memory for higher performance. Our results show MPI and Spark based iterative tasks benefit from high memory frequency and large number of channels. Among the configurable parameters, our results indicate that increasing the number of DRAM channels reduces DRAM power and improves the energy-efficiency.

Moreover, our result shows that changing the disk from HDD to SSD improves the performance of Spark, Hadoop, and MPI by 1.6x, 2.4x, and 3.3x respectively. However, I/O bandwidth caps the performance benefit of multicore CPU. Therefore, we developed an experimental equation to help designers to find the number of cores for which further increase does not enhance system performance noticeably.

Chapter 4

Memory Navigator for Modern Hardware in a Scale-out Environment

Today, more applications are moving to the cloud. Therefore, for cloud-scale servers, the increasing number of cores and applications sharing off-chip memory makes its bandwidth as well as capacity a critical shared resource. These trends suggest that it is important to understand the role of memory parameters such as capacity, number of channels, and operating frequency on performance of emerging class of applications in scale-out environment. The main contribution of this study is setting out a roadmap for memory configuration to maximize the performance cost ratio of cloud infrastructure.

4.1 Introduction

An empirical evaluation is important as it provides the community with reliable and accurate outcomes, which can be used to identify trends and guide optimization decisions.

To this goal, we first analyze various applications architectural characteristics. Based on the characterization results we classify applications into four different classes namely CPU intensive, IO intensive, Hybrid Memory-CPU intensive, and Hybrid Memory-IO intensive.

Based on this information we build a database and use it to drive an empirical performance model for each application class. Furthermore, we utilize IBM/SoftLayer TCO (total cost of ownership) calculator to drive a cost model for server platform in a scale-out environment such as cloud. The developed cost model takes into account the processor as well as memory parameters.

Based on the proposed predictive model, we present a novel methodology for selecting main memory parameters to maximize the performance per cost ratio of a given application in cloud. As the performance of memory subsystem depends on processor configuration, our methodology also navigates processor parameters as well as memory parameters (MeNa). MeNa is a three-stage methodology. It utilizes a fully connected Neural Network to classify a given application. After the classification, in the second stage, MeNa calculates the performance-cost sensitivity of application with respect to the server's parameters. In the third stage, MeNa solves a bounded knapsack problem using dynamic programming to find a configuration, which maximizes the performance per cost ratio.

Utilizing MeNa and based on the characterization results we make the following major observations:

1) Hybrid Memory-CPU intensive applications performance benefit noticeably from increasing the number of cores, low frequency core, low frequency memory, and large number of memory channels. 2) IO intensive applications are benefiting from small number of cores, high frequency cores, low frequency memory, and small number of memory channels. 3) Despite diverse range of frequency available in the memory market, increasing the memory frequency does not show to improve performance/cost ratio. 4) Increasing the number of memory channels improves the performance/cost ratio of hybrid Memory-CPU intensive applications. 5) Increasing the number of sockets increases the performance/cost of the system only if the number of cores per socket increases accordingly. 6) Increasing the capital cost of a server or a target budget set by a user does not always enhance in the performance/cost ratio of applications.

Table 4.1: Big Data Workloads

Workload	wordcount	sort	grep	pagerank	naïve bayes	kmeans
Domain	micro kernel	micro kernel	micro kernel	websearch	e-commerce	machine learning
Input type	text	data	text	data	data	graph
Input size	1.1 T	178.8G	1.1 T	16.8G	30.6G	112.2G
Framework	Hadoop, Spark	Hadoop, Spark	Hadoop, Spark	Hadoop, Spark	Hadoop, Spark	Hadoop, Spark
Suite	BigDataBench	BigDataBench	BigDataBench	BigDataBench	BigDataBench	BigDataBench

4.2 Experimental Setup

In this section, we present our experimental methodology and setup. We first present the studied applications and then introduce the studied big data software stacks. We will then describe our hardware platform and our experimental methodology.

Workloads

Diversity of applications is important for characterizing cloud platforms. Hence, we target three domains of applications from Big Data, multi-threaded programs, and CPU applications. For CPU and multithreaded applications we use SPEC CPU2006 [38] and PARSEC [14] benchmark suites, respectively. The studied big data applications are selected from BigDataBench suite [119], presented in table 4.1. BigDataBench has micro kernel applications as well as graph analytics and machine learning applications.

Hardware Platform

To have a comprehensive analysis of memory subsystem we used different SDRAM modules shown in table 4.2. All modules are from the same vendor. To build a cost model, we used IBM SoftLayer TCO Calculator, based on datacenter SJC01 (Located in San Jose, CA). A list of some of available processor types is presented in Table 4.3. For running the workloads, and monitoring the main memory, CPU, and disk behavior, we used a six-node server with detailed parameters for each node presented in table 4.4.

Table 4.2: Memory modules' part numbers

DDR3	4 GB	8 GB	16 GB	32 GB
1333 MHz	D51264J90S	KVR13R9D8/8	KVR13R9D4/16	—
1600 MHz	D51272K111S8	D1G72K111S	D2G72K111	—
1867 MHz	KVR18R13S8/4	D1G72L131	D2G72L131	KVR18L13Q4/32

Table 4.3: IBM SoftLayer bare metal servers

Processor type	#Socket	#Core	Core_freq	DRAM capacity	Disk bays	Net speed	Monthly charge
Xeon E3-1270	1	4	3.40 GHz	2 GB	2	2 Gbps	137 $
Xeon E5-2620	2	6	2.00 GHz	16 GB	12	10 Gbps	470 $
Xeon E5-2690	2	8	2.90 GHz	16 GB	12	10 Gbps	640 $
Xeon E7-4850	4	10	2.00 GHz	64 GB	6	10 Gbps	1602 $
Xeon E7-4890	4	15	2.80 GHz	128 GB	24	10 Gbps	2566 $

Architectural Behavior: We used Intel Performance Counter Monitor tool (PCM) [43] to understand memory and processor behavior. The performance counter data are collected for the entire run of each application. We collect OS-level performance information with DSTAT tool—a profiling tool for Linux based systems. Some of the metrics that we used for study are memory footprint, memory bandwidth, L2, and Last Level Cache (LLC) hits ratio, instruction per cycle (IPC), and core C0 state residency.

4.3 Characterization and Results

In a cloud platform, architecture and configuration of the server directly impacts its TCO and performance. The extent of this impact depends on the sensitivity of a cloud application to the architectural parameters and system configurations. Hence, we need to evaluate the performance sensitivity of our workloads to those parameters. Based on the level of sensitivity, we will classify the studied workloads. We then explore the relation between performance and TCO, and architectural configurations for each application class. This approach helps to formulize the relationship among configuration of cloud's platform, performance, and cost.

Table 4.4: Hardware Platform

Hardware Type	**Parameter**	**Value**
Motherboard	Model	Intel S2600CP2
CPU	Model	Intel Xeon E5-2650 v2
	# Core	8
	# Threads	16
	Base Frequency	2.6
	Turbo Frequency	3.4
	TDP	95
	L1 Cache	32 * 2 KB
	L2 Cache	256 KB
	L3 Cache	20 MB
	Memory Type Support	DDR3
		800/1000/1333/1600/1867
	Maximum Memory Bandwidth	59.7 GB/S
	Max Memory Channels supported	4
Disk (SSD)	Model	HyperX FURY
	Capacity	480 GB
	Speed	500 MB/S
Network Interface Card	Model	ST1000SPEXD4
	Speed	1000 Mbps

memory Analysis

We use IPC as a measure of application's performance. We consider the variation of workload's IPC, when we navigate memory and processor parameters, as an indicator for sensitivity of the application performance to those parameters.

1) Memory Sensitivity: Equation (3.4) expresses the memory bandwidth of the system as a function of number of channels, operating frequency and width. According to this equation, the maximum bandwidth that our platform supports is 59.7 GB/s (4 channel $\times$ 1.867 GHz $\times$ 8 Byte). Memory frequency is a characteristic of memory module and channel is the configuration of memory modules on the platform. The ability of using multiple channels effectively is decided by the support of memory controller. Because the focus of our study is on memory subsystem and its configuration, it is important to evaluate the sensitivity of our studied workloads to those parameters that are configurable, namely memory frequency, channels, and capacity. For our experiments, we used 3 sets of memory modules with different

(a). Relative IPC variation when increasing memory frequency from 1333 MHz to 1867 MHz

%IPC Variation	27	21	16	9	8	8	8	7	5	4
Workloads	canneal	facesim	libquantum	H-grep	milc	omnetpp	H-sort	gemsfdtd	bwaves	H-kmeans

(b). Relative IPC variation when increasing memory channel from 1 to 4

%IPC Variation	35	30	28	26	21	20	14	11	10	10	4
Workloads	S-sort	canneal	facesim	milc	H-grep	libquantum	Omnetpp	gemsfdtd	astar	H-kmeans	mcf

(c). Relative IPC variation when increasing memory capacity from 4 GB to 64 GB

%IPC Variation	25	22	18	10	8	7	7	7	5	5
Workloads	S-sort	S-wordcount	H-kmeans	S-nbayes	S-pagerank	H-sort	S-grep	H-grep	H-pagerank	H-wordcount

Figure 4.1: Memory sensitivity analysis

frequencies. A total of 22 different memory modules with a wide range of operating frequency, number of channels and capacity were selected based on their availability in the market for server class architectures. The memory modules frequency varies from 1333 MHz to 1867 MHz, number of channels ranges from 1 to 4, and their capacity is swept from 4 GB to 32 GB.

Figure 4.1 (a) and (b) show IPC variation when increasing memory frequency and memory channel, respectively, for a subset of studied applications. The interesting observation is that Spark-Sort workloads is the most sensitive application to the number of channels. Another interesting observation is that the sensitivity of most applications to memory channel is more than their sensitivity to memory frequency.

2) Bandwidth Sensitivity: Based on Equation (3.4) and the parameters of the studied memory modules reported in table 4.4, the minimum bandwidth that studied memory modules supports is 10.6 GB/s and the maximum bandwidth is 59.7 GB/s. Given that the studied workloads have different memory behavior and requirements, for off-chip memory bandwidth study we classify applications into memory intensive and non-intensive applications. The classification is done based on IPC variation as a function of memory bandwidth reported earlier in this section. Figure 4.2 presents the average utilization of off-chip bandwidth for each class of applications. According to this observation, memory intensive workloads use almost 4× more bandwidth than non-intensive workloads. This figure also shows that both memory intensive and non-intensive workloads cannot fully utilize the maximum available bandwidth. This implies the inefficiency of the modern server platforms when utilizing memory bandwidth. Our observation shows available memory bandwidth exceeds the needs of

all studied applications from various domains by approximately 10× and off-chip bandwidth is not a bottleneck for increasing the number of cores.

Figure 4.3 demonstrates the impact of core frequency on the average bandwidth usage of memory intensive workloads for two different memory configurations, one with maximum and the other with minimum memory bandwidth. The first configuration is a memory with one channel and memory frequency of 1333 MHz and the second is a four-channel memory and 1867 MHz frequency. Based on this figure, we observe that when the core frequency is low we can see both configurations can deliver required bandwidth for the workloads. However, by increasing the frequency, the bandwidth utilization of workloads is increasing. This is due to the fact that increasing processor frequency increases the number of memory request generates per unit of time. The bandwidth utilization gap between the two memory configurations increases when increasing the processor frequency. To show the impact of bandwidth utilization on performance, in Figure 4.4 we report the speedup in terms of relative execution time improvement comparing the two memory configurations. Increasing the core frequency up to 1.8 GHz does not bring performance advantage when using a higher bandwidth memory. However, it is only at frequency of 1.8 GHz and beyond where we observe a clear speedup gain using a high bandwidth memory. Therefore, the speedup gain when deciding memory configuration, is not only decided by the application type (memory intensive or not), but also by the maximum operating frequency of the core.

Figure 4.5 depicts which parameters of memory configuration help gaining more speed up when the core frequency is set to the highest; i.e. 2.6 GHz. The result shows memory configuration does not have any noticeable effect on the performance of memory non-intensive workloads, however memory frequency and number of channels impact the performance of memory intensive applications. In addition, the effect of memory frequency depends on the number of channels. By increasing the number of channels the influences of frequency on performance is reduced. We observe that increasing the number of channels from 2 to 4 does not improve the performance. This shows that the state-of-the- art server class memory controllers need to improve their management policy to effectively use 4 channels, otherwise 2

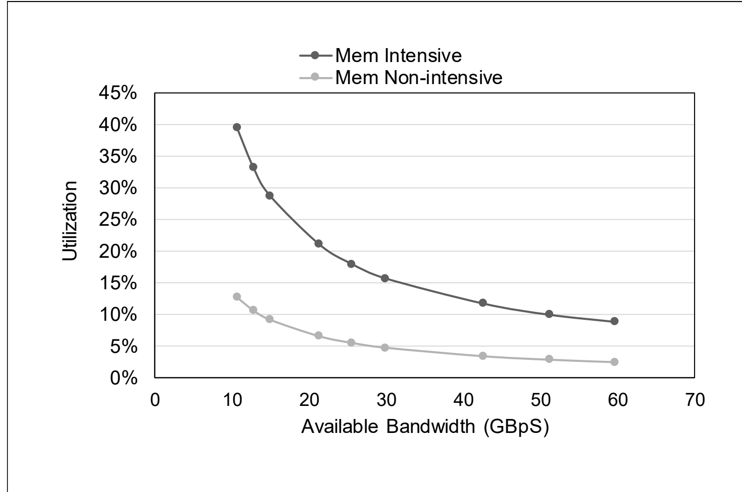

Figure 4.2: Bandwidth utilization

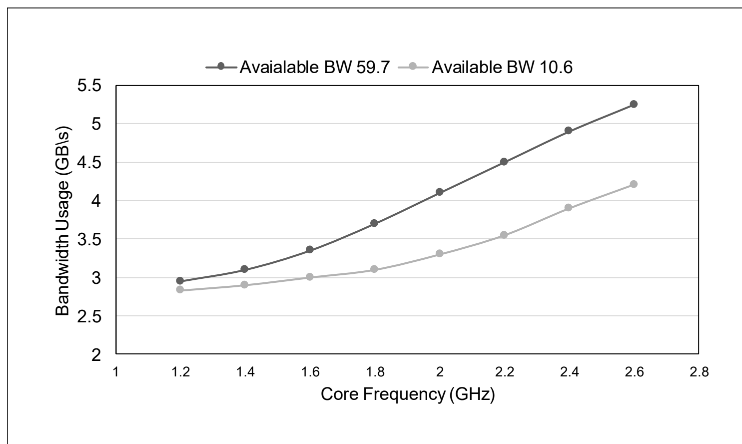

Figure 4.3: Bandwidth usage

channels is sufficient for a wide range of applications studied in this work. Memory controllers utilize a large fraction of the chip transistor budget and reducing the number of channels from 4 to 2 reduces the complexity of memory controller and therefore the entire processor, without sacrificing studied applications' performance.

3) Memory Capacity Sensitivity: Based on our results (not presented) we found that memory capacity and disk caching does not play a significant role for SPEC and PARSEC applications. However, for big data applications, due to their large input size, this is important to be investigated. To investigate the effect of memory capacity on the performance of Big Data applications, we run all workloads with 7 different memory capacities. During our

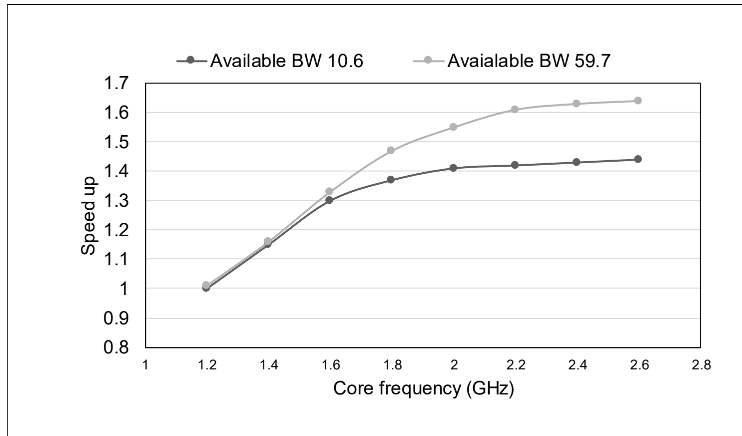

Figure 4.4: Speedup by memory

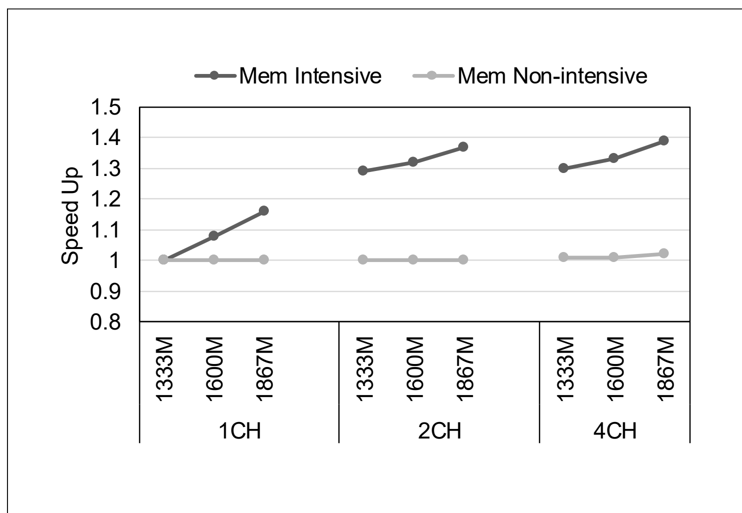

Figure 4.5: Speedup by CPU frequency

experiments, Spark workloads encountered an error when running on a 4GB memory capacity due to lack of memory space for the Java heap. Hence, experiments of Spark workloads are performed with at least 8 GB of memory. Sensitivity of Spark and Hadoop applications to the memory capacity has been presented in Figure 4.1 (c).

Microarchitectural Analysis

Figure 4.6 reports the microarchitectural analysis results for the two classes of studied applications. The first parameter to study is CPU stall. It is well known that front-end stall

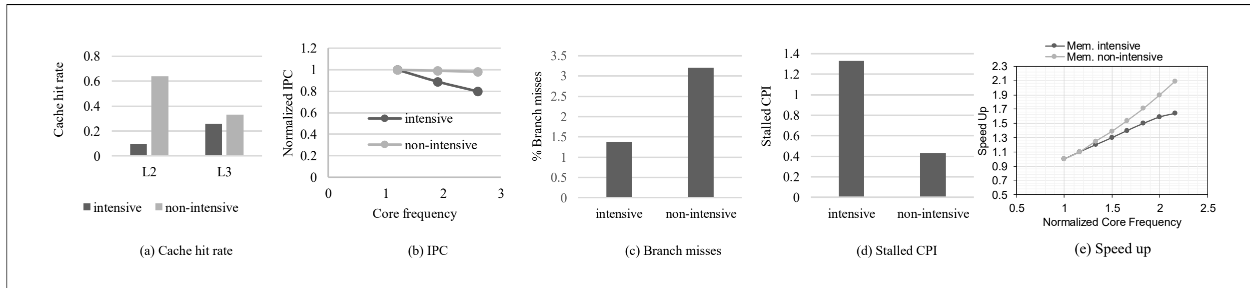

Figure 4.6: Workloads' microarchitectural behavior

directly incurs performance loss. Frontend stalls are also responsible for wasting power consumption. Figure 4.6 (d) shows stalled cycle per instruction for the two studied classes. Stalled CPI of memory sensitive workloads is almost 3 times more than nonsensitive workloads. Memory sensitive workloads suffer more from front-end stalls as the deep hierarchy of caches delays instruction–fetch and increases fetch penalty. While in general hardware prefetcher in modern multi-core processors are effective to improve performance of applications by reducing frequency of front-end stalls, for memory sensitive applications a noticeable front-end stall is still observed which indicates that a significant improvement is still needed for prefetchers.

Figure 4.6 (a) demonstrates the L2 and L3 cache hit rates. The main difference between memory intensive and nonintensive workloads is in L2 and L3 hit rates. Memory intensive applications show a very low L2 and L3 hit rate. As an example, Canneal, which is one of the most memory sensitive applications, has L2 and L3 hit rates of 0.02 and 0.03 respectively. However, for some applications, L3 can mask L2 misses. An example is Hadoop wordcount, with a 0.41 L2 hit rate, where its L3 hit rate is 0.44, enough to prevents Hadoop wordcount performance to suffer from low L2 hit rate.

Figure 4.6 (b) shows the average normalized IPC of memory intensive and non-intensive workloads for different core frequencies. The results show that increasing the core frequency reduces IPC of memory intensive applications. To show the effect of IPC reduction on the performance of the workloads in terms of execution time, we provide the average speed up of workloads in Figure 4.6 (e). The execution time results are normalized to the minimum

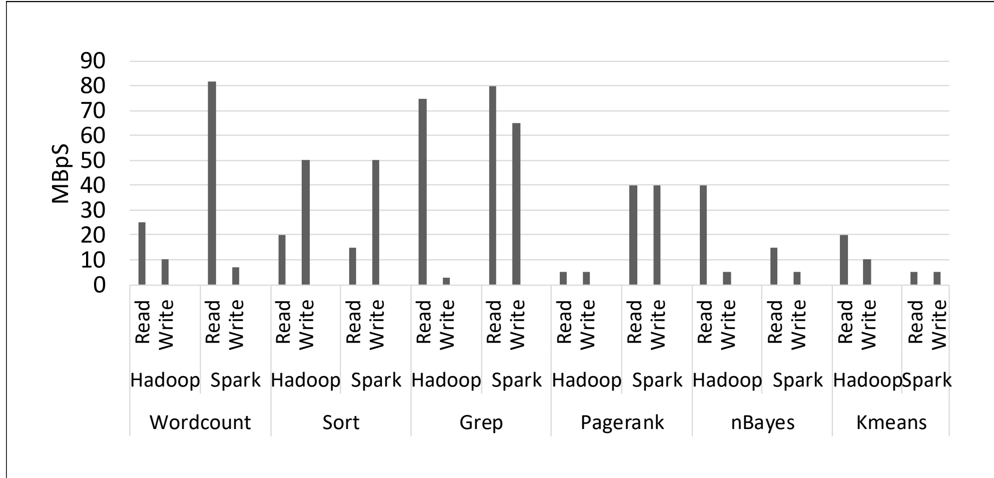

Figure 4.7: Disk access of Big Data applications

execution time of each application. The observation shows that memory non-intensive workloads speed up gain scale linearly with core frequency as their IPC impacted by only 2%, on average. For memory intensive workloads, the speed up curve falls below a linear curve, and even saturates when increasing the frequency beyond 2 GHz. As discussed earlier, increasing the bandwidth can mitigate this slightly. Increasing the available bandwidth from 10.6 GB to 59.7 only improves performance by 16%. Therefore, this is not an effective solution as the bottleneck exists in off chip memory access latency.

Based on Figure 4.7, we classify big data applications in two group of CPU-intensive and Disk-intensive class. Our decision criteria for this classification is based on the average Disk bandwidth usage. This Figure shows Spark wordcount, Spark grep, Spark PageRank, Hadoop Sort, Hadoop grep to be Disk-intensive while others to be CPU-intensive.

Workload classification. As the main goal of this paper is to study the combined impact of node architecture and cloud workload characteristics as well as performance/cost analysis, it is important to first classify those workloads. To this goal, we have explored the microarchitectural behavior of studied workloads to classify those workloads and find more insights. We divided workloads into two major groups of memory intensive and memory non-intensive. Each group of memory intensive and non-intensive applications will classify to two more Hybrid groups of I/O intensive and CPU intensive. This classification will help

us later to accurately formulate the relation of performance and application characteristics.

4.4 Performance and Cost Analysis

In this section, we formulate performance and cost analysis of different application classes in a scale-out environment. The first part of this section is devoted to formulating the total cost of ownership for different server configurations, using the cost offered by IBM/SoftLayer. We then develop equations to formulate the performance improvement of each application class with respect to the baseline configuration. These equations will be exploited by MeNa to select the most performance/cost efficient memory and CPU configuration for each class of application.

Cost Model

In this part, we analyze the parameters that are influencing the total cost of ownership (TCO) in a data center. Our goal is to establish a relationship between performance of studied applications, and the total cost of ownership when running these applications. We utilize EETCO [36] to drive a model for estimating TCO. The following five main factors determine the TCO in a data center:

- Datacenter Infrastructure Cost: the cost of acquisition of the datacenter building (real estate and development of building) and the power distribution and cooling equipment acquisition cost. The cost of the infrastructure is amortized over 10-20 years.

- Server Cost Expenses: the cost of acquiring the servers, which depreciates within 3-4 years.

- Networking Equipment Cost Expenses: the cost of acquiring the networking equipment, which depreciates within 4-5 years.

- Datacenter Operating Expenses: the cost of electricity for servers, networking equipment and cooling.
- Maintenance and Staff Expenses: the cost for repairs and the salaries of the personnel.

$$TCO = C_{infrastructure} + C_{server} + C_{network} +$$
$$C_{power} + C_{maintenance}$$

In the above equation, the first line represents the capital expenses (CAPEX) and the second line represents the operational expenses (OPEX). Given that the infrastructure, network, power, and maintenance costs are decided by the server configuration parameters, we can simplify the equation with $TCO = C_{server}(configuration)$ which indicates the total cost of ownership is a function of server configuration parameters.

The server price is determined as a function of server configuration as follow:

$$C_{server} = C_{processor} + C_{memory} + C_{disk} + C_{network}$$

The per-server costs include configurable DRAM, configurable processor, disk, and network costs. In this work, we do not consider configuring the network for performance optimization. Therefore, to establish a relationship between performances of applications as well as the price, we are simply treating the network cost as constant. We extracted the price data for 32 available server configurations in IBM SoftLayer bare metal servers. We used the regression technique to derive a cost equation for storage, memory, and processor.

All the below cost equations are the predicted charge that subscribers must pay in dollar for renting a bare metal server (on a monthly basis) on the IBM SoftLayer (data is collected in January 2017), which includes the power, cooling, and maintenance related costs of the server. The equation for price per server based on the server's processing configuration is as follow:

$$C_{processor} = \delta + \theta N_{socket} + \zeta Core + \kappa Frequency$$

Where $Frequency < 3.6GHz, Cache = 2.5MB/Core$, and $Core < 16$ per socket.

The values presented in table 4.5 are coefficients of parameters and eventually can be translated to the cost in dollar. We used MATLAB's regress library to fit our models with

Table 4.5: Values of processor cost's formula

Parameter	δ	θ	ζ	κ	R^2
Value	-353.5	208.1	31.4	54.9	0.82

the price data that we collected from IBM SoftLayer in January 2017. R-squared is a goodness-of-fit measure for linear regression models. This statistic indicates the amount of the variance in the dependent variable that the independent variables explain collectively. R-squared measures the strength of the relationship between our model and the dependent variable on a convenient [0,1] scale. 0 represents a model that does not explain any of the variation and 1 represents a model that explains all of the variation in the response variable around its mean.

For the cost of memory, we derived two different equations. The first considers the effect of memory frequency and the number of channels on the cost of each memory module. These parameters determine the available DRAM bandwidth for the processor. The maximum capacity of each available memory module is 32 GB. This is the maximum available DRAM module in the market (at the time of this research). In this work, we consider one module per DIMM.

$C_{module} = [(9 \times Capacity) \times (Mem.Frequency - 0.31)] - 5 \times N_{channel}$

where the memory frequency is in GHz and memory capacity is in GB.

Beyond 32 GB, the memory cost is estimated using the following equation:

$C_{memory} = (1.81 \times Capacity) + 364$

For the cost of storage, three types of storage are available such as SSD PCIe, SSD SATA, and HDD. In order to change the capacity or the bandwidth of storage, MeNa can aggregate multiple disks together. In this way, the cost of storage is as follow:

$C_{storage} = (N_{SSD-PCIe} \times Cost_{SSD-PCIE}) + (N_{SSD-SATA} \times Cost_{SSD-SATA}) + (N_{HDD} \times Cost_{HDD})$

Performance Model

Previously, we classified studied applications into 4 different classes. Based on our characterization and previous analysis, a set of performance equations is derived for each application class. These equations are developed using regression technique on a database collected through comprehensive experiments. We formulate those observations into regression-based equations to express performance of an application as a function of processor and memory configurations. Given the influence of both processor configuration and memory configuration on performance, we divide the performance gain equation into two parts; a part showing the performance gained by processor and another one showing the performance gained by the memory subsystem. The base configuration, which was used to account for performance gain is as follow: Number of sockets = 1, number of core per socket = 2, base frequency = 2 GHz, memory frequency = 1333 MHz, number of channels = 1, price = 73 $.

Table 4.6 shows the performance gain as a function of core count. For each class, we derived two different equations as the core frequency changes the behavior of applications. Similarly, Table 4.7 shows the performance gain by changing the core frequency.

Performance gain as a function of memory frequency and number of channels for various classes of applications is shown in Table 4.8. We only provide the equation for CPU-Memory intensive application class because other classes of applications do not gain noticeable performance benefit by increasing the memory frequency and the number of channels.

In addition, we derived the performance gain equation as a function of DRAM capacity as follows (only for Big Data applications, as the rest are not sensitive to DRAM capacity):

$Performance\ capacity = 0.0018\ capacity + 0.99$

The minimum capacity for Big Data application is also determined by the following equation:

$Minimum\ capacity = (footprint \times core)\ /\ 8$

In the next section, MeNa exploits these performance and cost equations to calculate performance/cost for a given user defined budget.

Table 4.6: Performance gain by increasing core count

App. class	Core frequency >2.8	Core frequency 2.8
Mem-CPU	Perf = 0.16 core + 0.67	Perf = 0.28 core + 0.42
CPU	Perf = 0.28 core + 0.48	Perf = 0.3 core + 0.38
IO	Perf = 0.1 core + 0.79	Perf = 0.14 core + 0.7
Mem-IO	Perf = -0.02 core + 1.04	Perf = -0.01 core + 1.43

Table 4.7: Performance gain by increasing core frequency

App. class	Core count >8	Core count 8
Mem-CPU	Perf = 0.04 freq + 0.96	Perf = 0.43 freq + 0.57
CPU	Perf = 0.25 freq + 0.75	Perf = 0.3 freq + 0.7
IO	Perf = 0.09 freq + 0.91	Perf = 0.26 freq + 0.74
Mem-IO	Perf = 0.03 freq + 0.93	Perf = 0.03 freq + 0.95

Table 4.8: Performance gain by memory frequency and channel

	High core and frequency	Low core and frequency
Memory frequency	Perf = 0.08 freq + 0.92	Perf = 0.03 freq + 0.96
Channel	Perf = 0.15 Ch + 0.89	Perf = 0.09 Ch + 0.97

4.5 Memory Navigator (MeNa)

In this section, we present our novel methodology for selecting and configuring DRAM system-level parameters in scale-out environment. Our methodology is based on the comprehensive analysis provided in previous sections.

Methodology

Figure 4.8 shows an overview of the proposed memory navigator. MeNa is a three-stage methodology, which navigates memory and CPU parameters to find the best performance/cost configuration for a given budget set by the user. In addition to memory parameters MeNa also navigates processor parameters as performance gain of memory subsystem is influenced by the interaction of both, as shown earlier in this chapter.

The first stage of MeNa is to determine the applications' behavior. Using the microarchi-

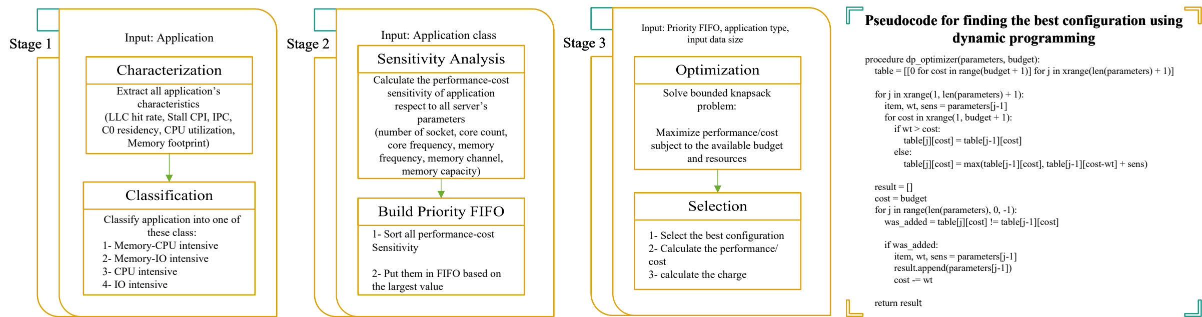

Figure 4.8: MeNa methodology overview

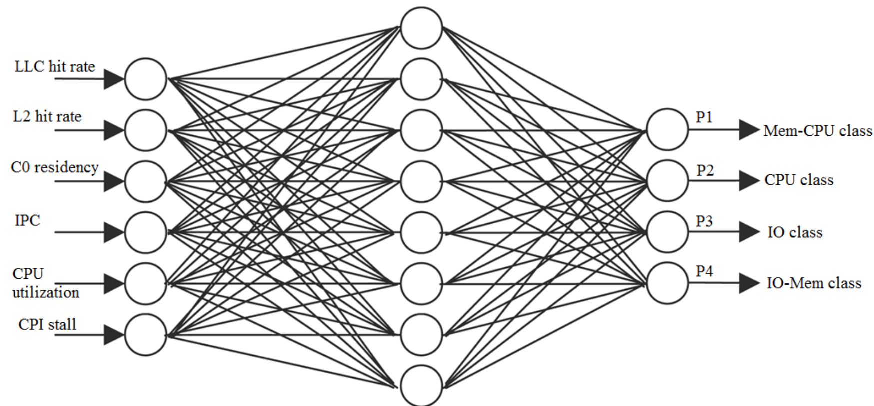

Figure 4.9: MeNa classifier (Neural Network)

tectural analysis presented earlier, MeNa classifies application into two main classes; memory intensive and memory non-intensive. Additionally, we divided applications to two more classes, namely CPU and I/O intensive, for more accurate performance estimate. Therefore, there are a total of four different classes as follow: 1) Mem-CPU intensive 2) Mem-IO intensive 3) CPU intensive 4) IO intensive. The classification is done on a three layer fully connected neural network trained by our training database. Figure 4.9 shows the first stage of MeNa. The neural network has 6 inputs and 4 outputs. Each output neuron stands for a class and it gives a probability between 0 and 1. Hence, a neuron with the highest value determines the class of application.

To identify the cost to increase performance by changing each server parameters, we define a quantity called performance-cost sensitivity. For example, the performancecost sensitivity to the number of cores per processor is defined as follow:

$Sens(Core)=((\partial Performance)/(\partial Core))/((\partial Cost)/(\partial Core))$

The second stage is to calculate this quantity with respect to the number of sockets per server, number of cores per processor, core frequency, memory frequency, number of memory channels, and the capacity of DRAM. The equations for performance are provided in tables 4.6, 4.7, 4.8, as well as cost equations. This quantity helps MeNa to set a priority for each parameter when allocating processor and memory resources. In this step, MeNa sorts all sensitivity results and based on the largest results it puts them into a FIFO. We refer to this as Priority FIFO.

In the third stage, MeNa determines the configuration to maximize the performance/cost while satisfying the subscriber budget. For this purpose, MeNa solves the following problem known as bounded knapsack by using dynamic programming:

Maximize $\Sigma_{i=1}^{n} Perf_i \times Conf_i$

Subject to $\Sigma_{i=1}^{n} Cost_i \times Conf_i \leq Budget$ and $min_i \leq Conf_i \leq max_i$

Where $Conf_i$ represents the number or the value of parameter i, min_i and max_i are the minimum and the maximum available resource for parameter i. Also, $Cost_i$ present the cost corresponding to $Conf_i$. Similarly, $Perf_i$ present the performance improvement corresponding to $Conf_i$.

The last step determines the final configuration and its corresponding performance/cost ratio as well as the corresponding cost using performance-cost equation.

Validation

To show how MeNa allows subscribers to intelligently search all server configuration space for finding the best parameters to maximize performance/cost while meeting the user specified budget, we validate MeNa against an oracle configuration identified through the brute force search. We apply the brute force search as follow: First, we select an application and set a budget. Then we find all configurations that achieve cost equal or smaller than the target budget. Finally, we run the application on those configurations and calculate its performance

Table 4.9: MeNa Validation

Application	Class	Budget	Configuration	#core	Core freq.	#Socket	Mem. Cap.	Mem. Freq.	#channel	Perf./Cost	Error
Spark Kmeans	Mem-CPU intensive	500$	Oracle	12	2.6	1	16	1333	2	0.977268	3.5%
			MeNa	13	2.8	1	12	1333	4	0.942737	
Hadoop Sort	IO intensive	180$	Oracle	4	2.8	1	4	1333	1	0.885733	10%
			MeNa	5	3	1	2	1333	1	0.796636	
Hadoop WordCount	CPU intensive	400$	Oracle	8	3	1	6	1333	1	1.010769	8.6%
			MeNa	10	2.8	1	8	1333	2	0.923691	

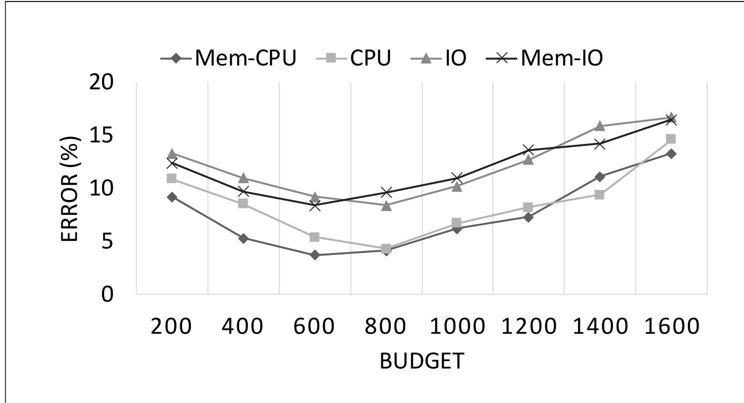

Figure 4.10: Average error for different class of applications and Budget

gain compared to the base configuration. By knowing the cost of each configuration, we calculate performance/cost for each configuration. The best configuration is referred as the oracle configuration. We then use MeNa to find the best configuration for the same application. Comparison between MeNa's outcome and oracle outcome shows that MeNa methodology can find the best configuration with 9% performance/cost error rate on, average for our training data sets. In the worst case, a 17% error is the price for avoiding an exhaustive brute-force search. The standard deviation of errors is 4%. Table 4.9 shows the error rate of MeNa compared to oracle configuration for three test applications from our training data sets.

Figure 4.10 shows the average performance/cost error rate of MeNa for various budgets. This result shows MeNa accuracy is higher for Mem-CPU class (8% error rate) while it has the lowest accuracy for I/O intensive class (12%). Moreover, MeNa is more accurate for mid-range target budget (between 500$ and 900$).

Table 4.10: Applications' features

Application	LLC hit	L2 hit	C0 residency	CPU util.	CPI stall	IPC	Class
S-Nweight	32	39	89.27	83	2.7	1.6	CPU-Mem
H-Terasoer	44	41	74.88	75	0.26	2	CPU
H-Scan	69	59	29.23	37	0.42	0.72	IO
S-srt	40	35	51.28	48	0.43	0.8	IO-Mem

Table 4.11: Configurations selected for Spark Nweight

Budget	250$	450$	700$	900$	1100$	1400$
#Core	7	13	13	6	12	16
Core_freq (GHz)	2.8	2.8	2.8	2.8	2.8	2.8
#Socket	1	1	2	4	4	4
Mem_cap (GB)	5	10	16	16	24	32
Mem-freq (MHz)	1333	1333	1333	1333	1333	1333
#channel	1	2	2	1	2	4
Perf/Cost	0.826133	0.765555	1.01344	0.792148	1.160709	1.53858
Error (%)	8.2	5.3	3.1	5.6	7.9	11.1

Table 4.12: Configurations selected for Hadoop Terasort

Budget	250$	450$	700$	900$	1100$	1400$
#Core	7	6	6	6	12	16
Core_freq (GHz)	2.8	2.8	2.8	2.8	2.8	2.8
#Socket	1	2	3	4	4	4
Mem_cap (GB)	3.5	6	9	12	24	32
Mem-freq (MHz)	1333	1333	1333	1333	1333	1333
#channel	1	1	1	1	1	1
Perf/Cost	1.119605	1.093411	1.077358	1.069507	1.589477	1.840536
Error (%)	9.8	7.5	5.9	6	9.4	10.8

Evaluation

In this section, we evaluate MeNa with unknown applications for various target budget. The selected applications are: Hadoop Terasort, Hadoop Scan, Spark sort, and Spark Nweight. Table 4.10 shows the features of each application and the class identified by MeNa. Tables 4.11, 4.12, 4.13, and 4.14 show the configurations selected by MeNa for the given budgets. We also report the performance/cost error rate of each application for each target budget. The results show that MeNa identifies a configuration with performance/cost of on average 11% close to an oracle configuration.

The evaluation of MeNa can also be used to derive architectural insights for server design-

Table 4.13: Configurations selected for Hadoop Scan

Budget	250$	450$	700$	900$	1100$	1400$
#Core	5	5	5	4	9	9
Core_freq (GHz)	3.6	3.6	3.6	3.6	3.6	3.6
#Socket	1	2	3	4	4	4
Mem_cap (GB)	2.5	5	7.5	8	18	18
Mem-freq (MHz)	1333	1333	1333	1333	1333	1333
#channel	1	1	1	1	1	1
Perf/Cost	1.046293	1.044781	1.044279	0.922948	1.449773	1.449773
Error (%)	14.6	14.8	12.2	13.5	15.4	17.2

Table 4.14: Configurations selected for Spark sort

Budget	250$	450$	700$	900$	1100$	1400$
#Core	3	3	2	2	4	4
Core_freq (GHz)	3.6	3.6	3.6	3.6	3.6	3.6
#Socket	1	2	3	4	4	4
Mem_cap (GB)	3	6	6	8	16	16
Mem-freq (MHz)	1333	1333	1333	1333	1333	1333
#channel	1	2	2	2	4	4
Perf/Cost	0.510734	0.455947	0.464989	0.445342	0.456145	0.364948
Error (%)	13.3	14.7	9.4	10.1	16	15.3

ers. For instance based on MeNa results we can see CPU intensive applications to demand large number of cores and low frequency processors while I/O intensive applications to require low number of core but high frequency processors. The result of I/O-Memory intensive application shows a very high performance/cost ratio since all options that can improve the performance are playing against each other. As a memory intensive application, we need large number of cores and high core frequency to take advantage of a fast memory subsystem. As an I/O intensive application, however, increasing the number of cores and core frequency exacerbate the I/O accesses and impacts application's performance. These findings have been corroborated by our characterization's result presented earlier in this chapter. Another observation is that increasing the memory frequency does not enhance performance/cost ratio even for memory intensive applications. On the other hand, increasing the number of memory channels improve the performance/cost ratio of memory intensive applications.

The trend in Figure 4.11 shows that, regardless of application class, scale-out approach cannot always enhance the performance/cost unless the scale-up solution is exploited. For

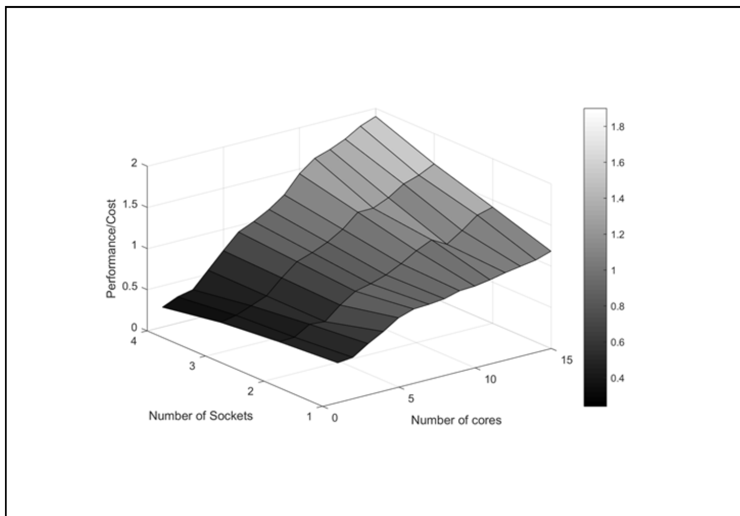

Figure 4.11: Average Performance/Cost correspond to the average number of core and socket

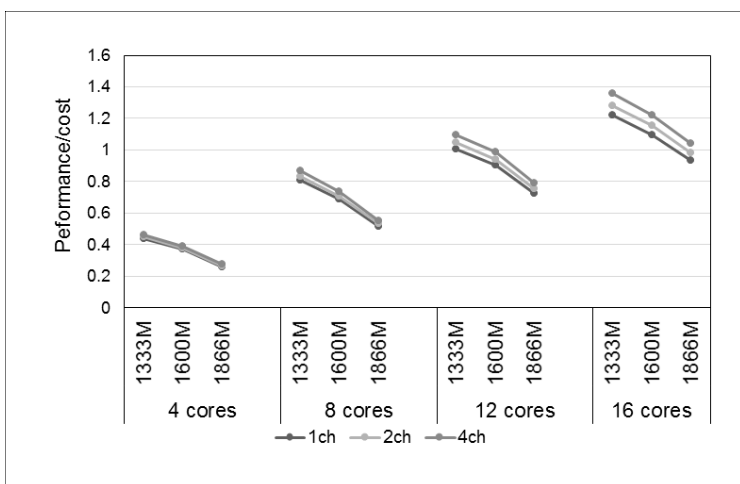

Figure 4.12: Effect of memory parameters on Performance/Cost

example, increasing the number of sockets, when the number of cores is low, reduces performance/cost. However, when the number of cores increases, performance/cost enhances by increasing the number of sockets. This means that the unseen cost that subscribers pay for a server such as for cooling, maintenance, and network, forces them to get the maximum utilization from their server. To show how memory frequency and number of channels affect the performance/cost of applications, we present the average results of our dataset for different number of cores in Figure 4.12. Figure 4.12 shows that increasing the memory frequency has almost no impact on improving the performance/cost ratio. On the other hand, increasing

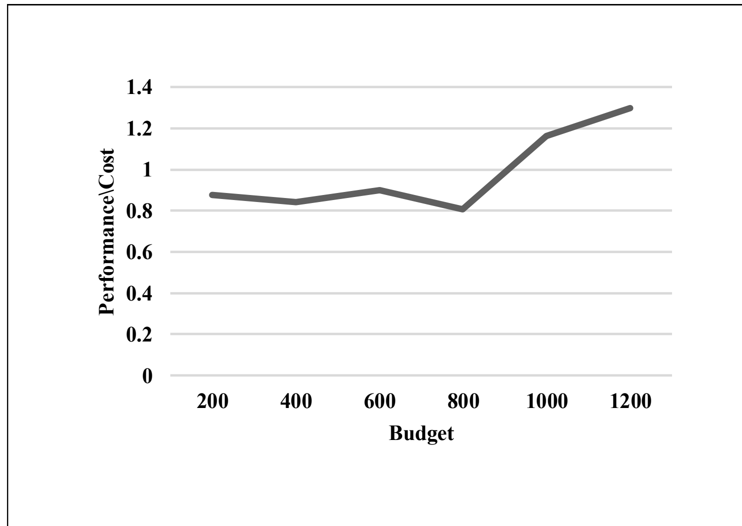

Figure 4.13: Average Performance/Cost correspond to Budget

the number of channel improves performance/cost ratio. However the improvement will be diminished if the number of cores is low. In fact, as long as the number of cores is high, enough pressure is being put on the memory subsystem, therefore results in enhancing the performance when the number of memory channels increases.

Another interesting observation is that allocating higher budget for an application does not necessary yield a better performance/cost, shown in Figure 4.13. This implies that there needs to be a method to enable subscribers to provision a rational budget for their application to get the max performance/cost benefits. MeNa proved that it is an answer for such urgent demand. MeNa methodology is architecture independent and therefore it can still be utilized for future technology.

4.6 Conclusion

Main memory performance is becoming an increasingly important factor contributing to overall system performance and the operational and capital costs. This particularly becomes important for server-class architectures as more applications are moving to the clouds. This suggests that it is important to understand the role of memory configuration parameters,

such as capacity, number of channels, and operating frequency, for performance and energy-optimization of emerging class of applications in scale-out environment. In response, this work addresses these challenges with a realsystem experimental setup. Our analysis reveals several interesting trends and provides key system and architectural insights on how memory configuration parameters must be tuned across various classes of applications to achieve high performance at low cost. To the best of our knowledge, this is the first work that provides a methodology for improving the performance/cost ratio of server class architectures in a scaleout environment while considering the memory parameter as well as processor parameters. We proposed a novel three-stage methodology to navigate memory parameter referred as MeNa. MeNa uses a fully connected Neural Network to characterize and classify applications. Based on the characterization results, we present experimentally derived models for estimating and predicting the impact of memory and processor parameters on capital and operational cost and performance of applications. MeNa uses those models to navigate memory and processor configuration parameters in order to find the best configuration to maximize performance/cost ratio for a given user defined budget. MeNa utilizes dynamic programming to solve bounded knapsack problem to achieve that goal. The validation results on our extensive database show MeNa to have 91% accuracy on average to estimate the performance/cost ratio compared to a brute force approach. MeNa enables subscribers to provision a rational budget for their application to get the max performance/cost in cloud environment. MeNa also reveals several interesting trends and provides key insights that can be leveraged by server designers for various optimization goals.

Chapter 5

Energy-Aware and NN-based RPS for IMC Application

In this chapter, we develop accurate and effective models to predict the resource requirements for each of the phases of IMC workloads at runtime, to determine an optimal energy-efficient configuration.

Neural networks has been used in several domains from wearable devices [84, 79, 80] to the cloud. We propose E-Net which leverages an artificial neural network to build a cross-platform energy-performance estimation model as well as an application's behavior predictor. Based on the developed predictive model and energy-performance estimator, E-Net uses an optimization engine to distinguish close-to-optimal configuration in order to minimize the Energy Delay Product (EDP) metric, which indicates the trade-off between energy and performance.

5.1 Introduction

In this work, we propose E-Net to overcome challenges discussed in section 1.1. With the aid of microarchitectural analysis, E-Net determines the characteristics of an application w.r.t.

the underlying hardware configuration. These application characteristics are employed for resource provisioning i.e., determining the best fitting architecture. E-Net is a system that proactively predicts the future behavior of running IMC applications using a time-series neural network (TSNN). It uses an artificial neural network (ANN) to dynamically estimate the Energy Delay Product (EDP) of an application for all available hardware resources, and provisions a near-optimal configuration that minimizes EDP while introducing low search overhead using a imperialist competitive algorithm.

The evaluation results show that the phase predictor of E-Net achieves 93% accuracy to correctly predict the future phase change of an arbitrary workload. On average, E-Net improves the EDP and the performance by 40% and 23% (up to 51%, and 36%), respectively, compared to a default scheduler. Such improvement in speedup is only achieved by efficient resource provisioning and without any software or framework tuning overheads. Based on the evaluation results, E-Net increases CPU utilization (average across all servers), DRAM bandwidth and memory capacity utilization by 74%, 49%, and 31%, respectively compared to default scheduler.

5.2 Background and Motivation

Energy is always important: Discussion in section 1.1 motivated us to propose an energy-aware resources provisioning system that allocates an appropriate amount of resources for not only performance benefits but also for consuming less power. The easiest and most effective way to decrease the power consumption and energy is reducing the voltage and frequency of the processor, however it increases the execution time and significantly degrades the performance. In this work we use EDP metric which considers both performance and power at the same time, and also is a popular metric in the computer architecture field, to measure the energy efficiency of the system. Equation (3.7) indicates how we calculated this metric.

Runtime changes in application's behavior: challenge to predict and act in

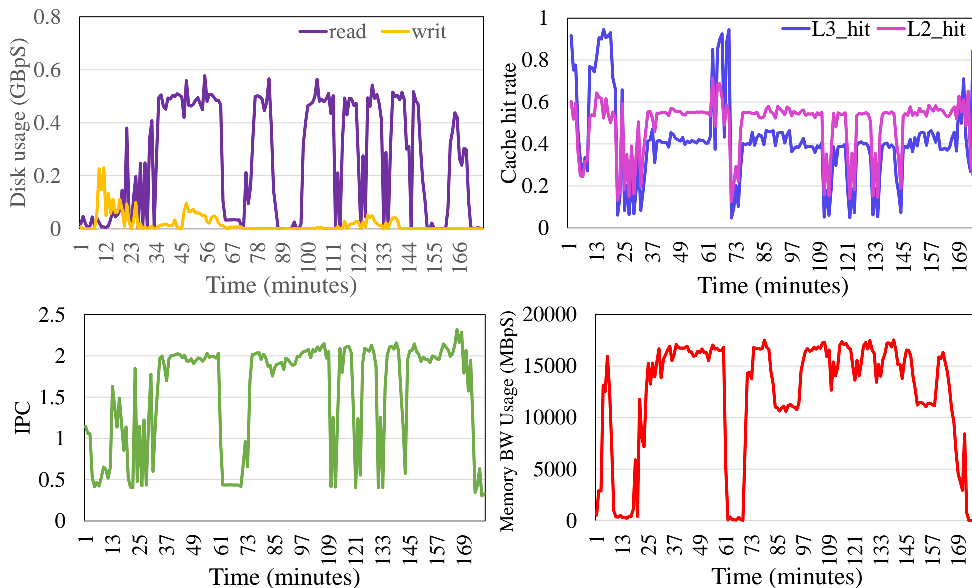

Figure 5.1: Example of application behavior and phase change

proactive manner: Each application faces various phases of execution, each with different memory and processing requirements. Figure 5.1 demonstrate the phase change of PageRank application during a part of its execution. Figure 5.2 shows the distribution of phases for a subset of studied workloads. As Figure 5.2 shows, Grep is a workload that is mostly I/O bound, Naive Bayes is memory and compute bound, CC (Connected Components) is mostly compute and I/O bound, and PageRank has all kinds of behavior, therefore each requiring different processing and memory resources to be allocated at different phases of runtime. Additionally, the results show the number of times the workloads behavior changes during the execution time. It is, therefore, necessary to identify those phases at runtime, to allocate resources accordingly.

While IMC applications' behavior changes multiple times during execution, their transitions are predictable. E-companies typically run IMC programs repeatedly for a long period of time with different contents but similar input dataset size [93]. We refer to this type of programs as periodic (daily, weekly, and etc.) long jobs. This dynamically varying characteristic motivates to develop a model that can proactively predict application behavior

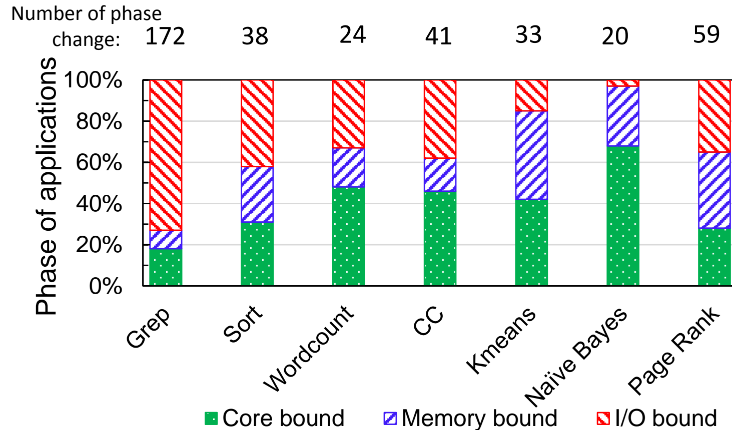

Figure 5.2: Average phase distribution of Spark workloads

changes online and allocate appropriate processing and computing resources, accordingly.

Model must account for both hardware and software characteristics: Prior works mostly attempt to tune software parameters on a single server configuration [59]. However, given the heterogeneity and diversity of processing and memory architectures in the cloud, it is important to account for hardware configurations, as well. In addition, while prior works mostly tune parameters for better performance [130, 127, 140], tuning for optimal energy is also important and challenging. The main challenge is how to tune the parameters for both power and performance, as a configuration that is optimized for power in most cases deliver sub-optimal performance and vice-versa. To respond to this conflicting optimization goal, i.e. tuning for power and performance simultaneously using energy-delay product metrics (EDP), in this work we use an artificial neural network (ANN) for estimating the energy consumption of each application corresponding to each server platform. To accurately estimate the performance and power, the deployed ANN takes the target server's configuration as well as the application's microarchitectural signature as inputs to account for both hardware and software features. In a real-world cloud environment, an IMC program can run repeatedly multiple times with similar size of input datasets, while the underlying hardware can be different because of the virtual environment. This motivates our work to seek new modeling solution to provision hardware resources for a given IMC program for

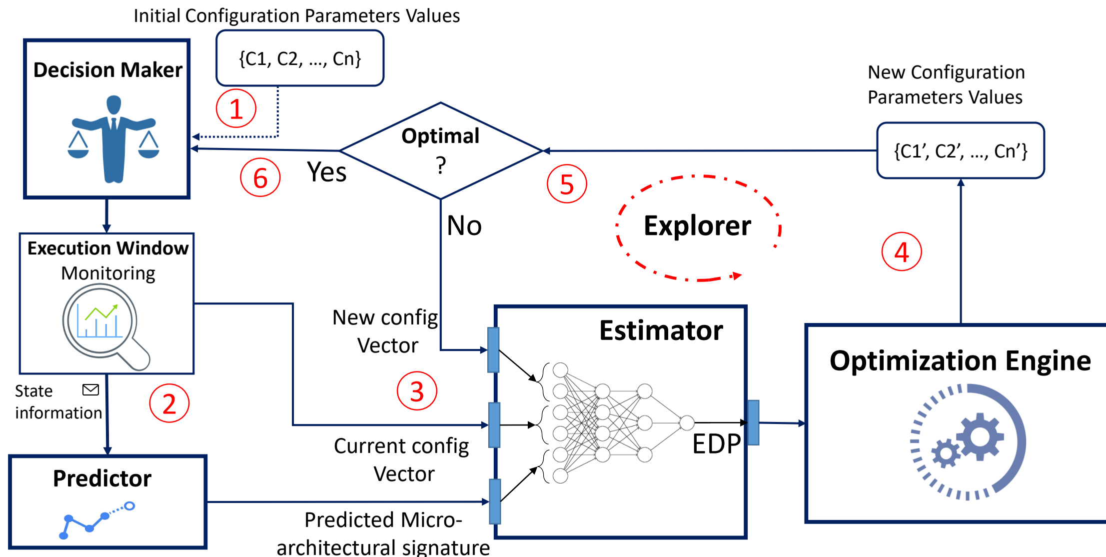

Figure 5.3: E-Net overview

energy optimization goal. E-Net is a comprehensive system that leverages machine learning technique (e.g. neural networks) to address all the aforementioned challenges.

5.3 E-Net Methodology

E-Net is a configuration tuning methodology that automatically adjusts the hardware configuration assigned to a Virtual Machine (VM) in a proactive manner in order to dynamically optimize the energy efficiency of a given IMC program on a given heterogeneous cluster of servers. E-Net consists of four major components: predictor, estimator, explorer, and decision maker. Following subsections elaborate the details of each component.

System Overview

Figure 5.3 illustrates the block diagram of E-Net. E-Net server maintains a database of per-host state and updates it on each interval. E-Net predicts the next phase of an application and its microarchitectural signature based on the current and previous states. Given the predicted signature and corresponding server configuration, E-Net estimates the application's

energy consumption and performance in term of the EDP metric. Next, E-Net searches for the best platform and configuration that minimize the EDP for a given IMC application. The searching component (explorer) automatically searches for the configuration that achieves minimal EDP. Overall, the estimator component of E-net relies on the results of the predictor, and the explorer component selects the best configuration from the outcome of the estimator.

Online Monitoring

In order to continuously monitor each host's state, a monitoring agent runs on each host. Monitoring agent extracts architectural information and resource utilization of application during each execution window and reports it periodically to the E-Net server. We define the window as a sufficient amount of time that a VM on a particular physical server can be maintained without migration. In this study, we set the window size to 3 minutes. The architectural information is collected through the Intel Performance Counter Monitor tool (PCM) [43] to capture the memory and processor behavior. Although PCM reports the power consumption of CPU and memory, we installed WattsUp power meter on each server to measure the whole power consumption of the server. To capture application behavior we studied 17 features in the following broad categories:

(a) **Memory related:** Available virtual, physical, and shared memory, the cache and buffer space, and memory bandwidth utilization.

(b) **Disk related:** Ratio of free to total disk space, and storage bandwidth utilization.

(c) **Network related:** Bytes sent and received.

(d) **CPU related:** L2, and Last Level Cache (LLC) hits ratio, instruction per cycle (IPC), core C0 state residency, and CPU idle, system, user time.

We consider this information as an architectural signature of an application.

Predictor

As shown previously, an IMC application has various phases during its execution time. We define a phase as a period of time that application's behavior remains unchanged. Here, the application behavior is interpreted as a way that the application utilizes different server resources and exhibit various architectural features. As the proposed E-Net needs to be proactive in order to determine an optimal server configuration at runtime, it must be able to act before a significant change occurs in the behavior of application which influences the performance and power of the system. In order to act in a proactive manner, E-Net is equipped with a phase predictor to predict the future phase and its corresponding architectural signature of the application. For this purpose, we employed Time Series Neural Network (TSNN) technique and also traditional ML models such as hidden Markov model, and K nearest neighborhood, and support vector machine (SVM). Each of these techniques has its own trade-offs in terms of accuracy and complexity. However, we eventually decided to employ TSNN. The detailed explanation of this decision is explained later in section 5.5. The accurate prediction is important as all other steps of E-Net depends on the predicted architectural signature of application to assign an appropriate resource to the VM in advance, before the performance is changed (degraded) or resource utilization drops.

SVM

We also explore the Support Vector Machine to model the data, due to the benefits of relatively lower complexity and similar performance. SVM analysis is a popular ML tool for nonlinear functions. SVM is considered a non-parametric technique because it relies on kernel functions. Some problems cannot adequately be described using a linear model. In such a case, the Lagrange dual formulation in SVM allows the technique to be extended to nonlinear functions.

Hidden Markov model

The hidden Markov model (HMM) is another eager technique employed for effective prediction. The HMM is used extensively for performance modeling and performance-prediction analysis, where the HMM can predict the future state of a target system based on its current state. In reality, as the relationship between the observed time and the observed state is not one to one, a group of probability distributions for two stochastic processes are involved, called the HMM. In an HMM, the states are not observable, but when we visit a state, an observation is recorded that is a probabilistic function of the state.

K nearest neighbors (KNN) regression

KNN [105] is a lazy learning technique that does not require training. Suppose the dataset has m samples that each sample x_i is described by n input variables and an output variable y_i such as $x_i = \{x_{i1}, ..., x_{in} | y_i\}$. The goal is to learn a mapping function F: x to y known as a regression function that captures and models the relationship between input variables x and an output variable y. The KNN regression estimates the function by taking a local average of the dataset. Locality is defined in terms of the k samples nearest to the estimation sample. As the performance of KNN algorithm strongly depends on the parameter k, finding the best values of k is essential. A large k value decreases the effect of noise and minimizes the prediction losses. However, a small k value allows simple implementation and efficient queries.

Time series neural network

Time series neural network or TSNN [134] is an eager learning technique. The training of TSNN is done offline by our database. The time series neural network module is based on a nonlinear autoregressive network with exogenous inputs network.

Equation (5.1) is deployed to predict the next architectural signature of IMC application.

$$Y(t) = F(Y(t-1), Y(t-2), ..., Y(t-n)) \tag{5.1}$$

For the prediction of each architectural feature, we used one TSNN. The output of predictor is a vector (Sign) and is fed to the EDP estimator. In Equation (7.1), a_i denotes each architectural feature.

$$Sign = \{a_1, a_2, ..., a_{17}\} \tag{5.2}$$

EDP Estimator

It has been shown in prior works that using analytical modeling techniques [39, 40] is not an effective solution for predicting the performance and power since these methods suffer from low accuracy caused by over-simplified assumptions [12]. In order to model the EDP of each application on different platforms, in this work we used Artificial Neural Networks (ANN). We will show that ANN-based estimator is an effective, accurate, and fast approach to model the EDP of applications with respect to their microarchitectural behavior.

ANN attempts to estimate the performance and power consumption of monitored application for a target server specified by server configuration inputs. The ANN has three sets of inputs (total 37 inputs). The first set of inputs is the current parameters of the server. The second set of inputs is the architectural signature of application coming from the predictor which is called signature inputs. The third set is the proposed configuration parameters of the server platform referred to configuration inputs The configuration inputs vector is as follow:

$$Conf = \{c_1, c_2, ..., c_{10}\} \tag{5.3}$$

where $Conf$ is the configuration vector and c_i is the value of the ith configuration parameter (number of sockets, number of cores, core frequency, cache size, memory capacity, memory frequency, number of memory channel, storage capacity, storage speed, network bandwidth).

Next, the ANN model estimates the EDP based on $Sign$ and $Conf$. The EDP model is described by:

$$EDP = f(Conf, Sign) \tag{5.4}$$

Note that $f(Conf, Sign)$ is a data model, which means there is no equation to describe it.

Once the EDP of application is estimated, the E vector is constructed to store the EDP and the corresponding configuration. The E vector is defined as follows:

$$E = \{EDP, Conf\} \tag{5.5}$$

where E is an input for the explorer.

Explorer

The purpose of Explorer is to find the best configuration that minimizes the EDP. Therefore, it is an optimization problem which the optimization variables are server's configuration and the cost function is EDP. In order to solve this problem, Explorer uses an optimization engine (OE). In this work, we explore the effectiveness of popular optimization algorithms such as genetic algorithm (GA) [64], Imperialist competitive algorithm (ICA) [52], Discrete Particle Swarm Optimization (PSO) [121], and Bat algorithm [92]. The results (presented in section 5.5) show that the ICA achieved better convergence to the global optimum, thereby we use it in our optimization engine. There exists other techniques for navigating complex configuration spaces, e.g., recursive random search [129], and pattern search [108]. Random recursive search has shown to be ineffective since it is more likely to be locked in local optima, preventing the system to achieve the maximum efficiency. Pattern search typically suffers from slow local (asymptotic) convergence rates [108].

Algorithm 1 describes the configuration search procedure which consists of six steps, as shown in Figure 5.3. In step 1, we run the application on a random platform (random configuration) and start monitoring its microarchitectural behavior for N execution windows ($N = 12$ for experiments conducted in this research). Next, we extract the architectural information to predict the next phase of the application using TSNN in step 2. In step 3, we input the initial values of the configuration parameters and architectural signature of

Algorithm 1 Explorer functionality

Input: Architectural signature of application, List of available servers
Output: An optimal configuration
Result: Minimized EDP

```
1 Optimal_EDP = ∞
2 Conf ← Get_Random(List of servers)
3 Sign ← Architectural Signature
4 E ← ANN_Estimate(Conf,Sign)
5 while iteration ≤ Threshold do
6     Opimal_EDP← OE_Assimilation(E_EDP,Opimal_EDP)
7     Conf' ← OE_Im_compet.(OE_Revolu.(E_Conf))
8     E ← ANN_Estimate(Conf',Sign)
9 return E_Conf
```

the next phase of application to the EDP estimator. In step 4, we pass the estimated EDP and configuration parameter values to the OE. Note that the configuration parameters are randomly selected. OE is based on ICA. Therefore, we define an array of values of a candidate solution as a Colony. The OE then performs a number of operations such Assimilation (Colonies move towards imperialist states in different in directions), Revolution (Random changes occur in the characteristics of some countries), and Imperialistic competition (All imperialists compete to take possession of colonies of each other). In this way, a new set of configuration parameter values is generated. In step 5, these configuration parameter values are fed to the estimator to estimate a new EDP value and the EDP is passed to the OE again to check if the stop condition is satisfied or not. Steps from 3 to 5 are repeated for a number of times until the optimum configuration is found. The next step (6) includes sending the optimal configuration to the decision maker. All steps are repeated for the next window of operation until the execution of the IMC program finishes.

Decision Maker

After finding the optimum or close the optimal configuration, E-Net manager follows a methodology which divides the resource allocation problem into smaller sub-problems using a hierarchical approach. It demonstrates which actions can be executed in what level for efficient resource allocation of scale-out infrastructures. In general, the following reallocation actions can be considered:

- 1) Increase CPU frequency
- 2) Decrease CPU frequency
- 3) Increase CPU core (number of cores of a host allocated to a VM)
- 4) Decrease CPU core
- 5) Add allocated storage.
- 6) Remove allocated storage.
- 7) Increase memory capacity.
- 8) Decrease memory capacity.
- 9) Migrate VM to a different host.

Note that in all experiments there is an underlying mechanism (Xen Hypervisor) that performs the live migration. Decision maker migrates VM when the migration latency is less than half of window's time and also when it is necessary.

Changing the CPU frequency is performed by DVFS. The first escalation level ("change VM configuration") works locally on a host and attempts to change the host resources. This operation is performed by hot-(un)plugging of resources (memory and cores). For example, mounting storage or adding memory to the VM is more lightweight than migrating VMs. If there is no appropriate VM available in level 1, the second escalation level is called, where

the decision maker creates a new VM on an appropriate host or migrates the VM to a host that has enough available resources.

Migration consideration: The manager must take into account the Xen live migration time to decide whether to migrate the VM or not. Therefore, we adopt the performance model of Xen live migration proposed in [82] to predict the time that takes to migrate a VM from node A to node B and resume the job.

Performance modeling of live migration involves three main factors: the size of VM memory (V_{Mem}), the memory dirtying rate (D), and network transmission rate (J). Live VM migration achieves negligible application downtime by iteratively pre-copying the pages dirtied at the previous round of transmission. Xen provides the ability to track memory accesses of guest VMs using a mechanism referred to shadow page tables. The shadow page tables are maintained by the hypervisor and translated from guest page tables on demand. In this way, the hypervisor is able to trap all memory updates within a VM and maintains a bitmap to mark the dirty pages. As VM live migration also works in shadow paging model, we can measure a VMs memory dirtying rate incidentally before the pre-migration phase. The data transmission rate for each round is determined by adding a constant increment to the previous rounds memory dirtying rate (D) where the constant variable and its default value is empirically set at 100 Mb/s. Let λ denote the ratio of D to J. Then we have the migration latency:

$$T_{mig} = \sum_{i=0}^{n} T_i = \frac{V_{mem}}{J} . \frac{1 - \lambda^{n+1}}{1 - \lambda} \tag{5.6}$$

5.4 Implementation

In this section, we present our experimental system configurations and the setup. We first present the E-Net prototype. We then describe our hardware platform which runs the E-Net cluster. Lastly, the frameworks and the workloads for evaluating E-Net are introduced.

E-Net prototype

E-Net prototype is written in Java with about 3K LOC and is integrated into Apache CloudStack [106] which is an open-source cloud management software for running a private cloud infrastructure. It has enterprise-class support for scaling out VMs on XenServer hosts and controls the XenServer host instances using the Java bindings of the XenServer Management API. XenServer is a Linux distribution that is based on the Xen hypervisor and running Linux Ubuntu 16.04 LTS in our work. The API designed for E-Net includes functions to express the type of submitted workloads, and functions to check job status, revoke it, or update the constraints. E-Net currently can manage IMC frameworks such as Spark and Flink within VM with no need to change the applications code.

Figure 5.4 shows the overview of E-Net cluster. We implemented the monitoring agent in Dom-0, which is a privileged VM in Xen hypervisor. E-Net server maintains a database of per-host state and updates it on each interval. Each record of the database contains the information of each workload on each server and each record size is roughly 1080 bytes. Neural Networks used in this work are implemented in MATLAB [6]. To use them in E-Net framework, we used MATLAB Compiler to compile MATLAB code into a shared library to be able to call it from our Java code. To implement genetic algorithm, we used *Jenetics* Programming Library [5]. All components of E-Net (predictor, estimator, explorer, and decision maker) are running on the server side and they do not have any impact on the execution of IMC applications.

Cluster Setup

We implemented E-Net as a centralized resource provisioning server on Argo cluster located at George Mason University with a total of 20 hosts. The local cluster includes servers of 12 different configurations shown in Table 5.1. As seen from the table, we also show how many servers of each type we can use. Note that these configurations range from high-end Xeon servers to low-end ones. There is a wide range of core counts, clock frequencies, storage type,

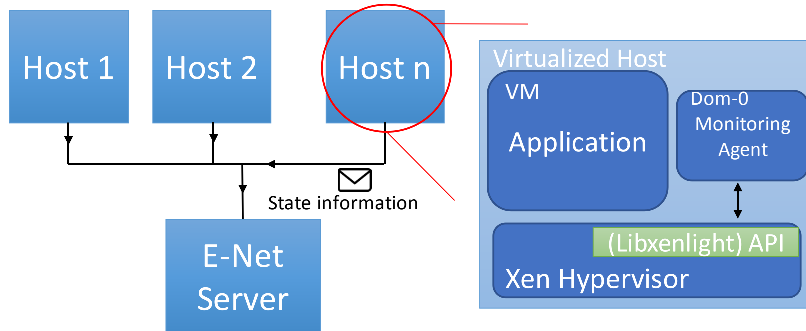

Figure 5.4: Overview of E-Net cluster

and memory capacities and bandwidth in our experimented cluster. All servers are equipped with at least one high speed 1 Gbps network interface card.

Table 5.1: Detailed information of local cluster

Server (Xeon)	Freq. (GHz)	Socket	Core	Cache (MB)	Mem. (GB)	Storage	Server type	Count
E5-4669 V4	2.2	4	22	55	96	SSD PCIe	HPC	1
E5-4667 V4	2.2	4	18	45	64	SSD SATA	HPC	1
E5-4650 V4	2.2	4	14	35	32	SSD SATA	HPC	1
E5-2690 V4	2.6	2	14	35	512	SSD / HDD	Memory opt.	2
E5-2650 V4	2.2	2	12	30	256	SSD / HDD	Memory opt.	2
E5-2667 V4	3.2	2	8	25	32	SSD PCIe	I/O opt.	2
E5-2643 V4	3.4	1	6	20	32	SSD PCIe	I/O opt.	2
E5-2660 V2	2.2	2	10	25	16	HDD	General purp.	3
E5-2650 V2	2.6	2	8	20	16	HDD	General purp.	3
E5-1630 V4	3.7	1	4	10	8	HDD	Power opt.	1
E5-1680 V4	3.4	1	8	20	12	HDD	Power opt.	1
E3-1270 V6	3.8	1	4	8	8	HDD	Power opt.	1

Representative Applications

Apache Flink [20] is the latest framework to the list of open-source frameworks focused on Big Data Analytics that are trying to replace Hadoop and Spark. Flink is built for in-memory processing of batch data, like Spark. This model outperforms Spark when repeated

passes need to be made on the same data. This makes Flink an ideal candidate for machine learning and other use cases that require adaptive learning, self-learning networks, and etc.

For training of E-Net neural networks' models, we target various domains of data-intensive workloads such as microkernels (Wordcount, Sort), graph analytics (Nweight, Connected Components), machine learning (Kmeans, Naive Bayes), E-commerce (PageRank), and social networks (Grep). In total 8 workloads for each Spark and Flink framework are experimented from BigDataBench 4.0 [119] in which the size of the input is in the range of 200 GB up to 4 TB. The selected workloads have different characteristics such as high-level data graph and different input/output ratios. Some of them have unstructured data types and some others are graph based. Also, these workloads are popular in research and are widely used for the demonstration of techniques. In our study, we used Spark version 2.1.0 in conjunction with Scala 2.11, and Flink version 1.3.3.

Offline Benchmarking and Training

Figure 5.5 illustrates the overview of workloads benchmarking and models training divided into two phases: Collecting and Training. In the collecting phase, we perform these experiments: we generate a number of configurations, automatically run IMC programs with the generated configurations, and collect the architectural information, the execution times, and power consumption of the running workload. In the training phase, we use the benchmarking database to train the ANN for the EDP estimation as a function of the high dimensional server configuration parameters and application's architectural signature, and also to train TSNN for the future architectural signature prediction.

Data Collecting: For a given IMC program, we create a database of power and performance as follows:

1) We use a configuration generator to create a configuration which is a vector containing *10* configuration parameter values for each experiment ($Conf_i$).

2) We then run the IMC program with its input dataset on a server with $Conf_i$ configu-

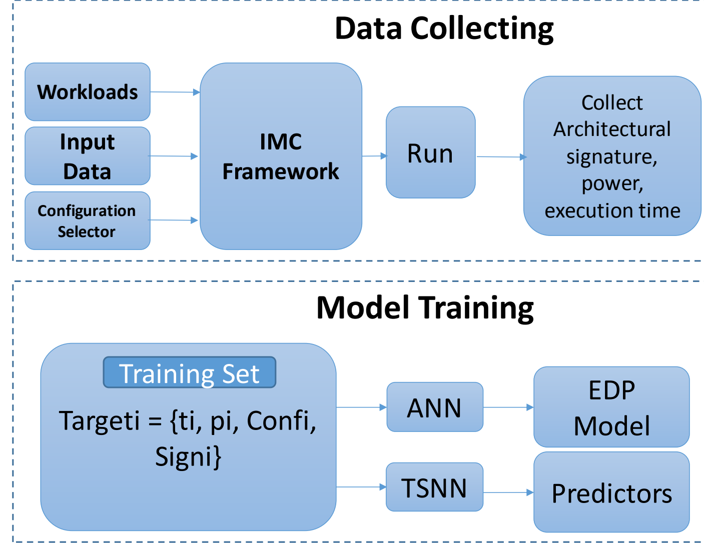

Figure 5.5: Overview of benchmarking and training

ration.

3) During the execution, we monitor the application and extract its architectural signature ($Sign_i$) for each execution window. Eventually, we store the information in a vector as a time series data.

4) After the execution of the program, we create a vector to save the execution time (t) and power consumption (p) with the corresponding server configuration and application signature. Given the estimated values, we calculate EDP as follow:

$$EDP = (t)^2 \times p \tag{5.7}$$

$$Targeti = \{EDP_i, t_i, p_i, Conf_i, Sign_i\} \tag{5.8}$$

Model training: We use $Target_i$ vector as a training set for building the EDP estimator. We use EDP_i as a target data and $Conf_i$ and $Sign_i$ as input data for the training of ANNs. In this work, we used a ten-layer fully connected ANN with the following architecture [37, 105, 80, 80, 75, 60, 65, 45, 15, 1] for EDP estimation. The number of neurons for the hidden layer is decided through Grid Search [104] to reach the highest possible accuracy. Moreover,

we use the time series data to train different ML models for future phase prediction of IMC applications.

To determine the best set of hyperparameters, we employed Grid search. 17 predictors from each type of ML model are used that each of them is trained for one metric. Each model has 12 inputs and one output. Each input shows a single timestamp sample and the output shows the predicted value. When a job has just started and does not have the full 12 history measurements, we use the current data for all previous stamps.

5.5 Results and Evaluation

Overhead

The collecting data incurs the highest cost, 31.7 hours on average and up to 39.2 hours for each workload. Model training of each TSNN took 48 minutes on average and ANN for EDP estimation took 393 minutes. While training phase is very long, it is a one-time cost and is still attractive compared to manual configuration. It is important to note that the target of E-Net is the iterative applications which repeatedly run in data centers for months or even longer. In this usage scenario, this high one-time cost is amortized with a very large number of runs. Therefore, the additional overhead per run is very low. Moreover, when a new application or a server comes, the whole data collection procedure is not required to be redone. In this case, when the predictor finds a new behavior, we just save that trace and report it to manager. After a while, when a collection of new traces has been collected, new traces will be added to the database to only retrain the predictor and estimator. In this way, we can easily update models with low overhead. Searching optimal configuration using OE is performed in few seconds (1.2 s) for 65 iterations. This processing time is negligible compared to the window's size (3 minutes).

To show the results of scheduling decisions made by decision maker, we report the the number of migrations and VM resource changes for studied applications in Table 5.2. The

results show each VM migrated 10 times on average. The average migration latency is 89.8 seconds. Moreover, the ratio of total migration time to total execution time is around 5.3% which is acceptable. Based on the results, the downtime of each migration is around 370 ms on average.

Table 5.2: Information of scheduling decisions (Average results for each Virtual Machine)

Applications	WC	Sort	NW	CC	KM	NB	PR	Grep	Average
Number of phase change	24	38	65	41	33	20	59	172	56.5
Number of change in VM configuration	22	33	56	37	28	18	53	151	49.7
Number of VM migration	5	6	10	8	7	4	11	32	10.3
Average VM migration latency (Second)	95	81	103	92	94	87	91	76	89.8
Standard deviation of migration latency	17	16	31	10	24	16	12	9	16.8
Average down time (millisecond)	378	264	571	342	389	355	367	294	370
Total migration time to total execution time (%)	6.2%	4.0%	5.1%	5.7%	6.4%	5.6%	5.4%	4.6%	5.3%

Accuracy

In this work, we used 5-fold cross-validation to evaluate the accuracy and validity of the predictors and estimator. In order to avoid over-fitting and enhance the accuracy, we deploy dropout technique with probability of 0.2.

To evaluate the accuracy of the generated predictors and the estimator, we use RMSE (Root Mean Squared Error) equation.

$$RelativeRMSE = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(\frac{p_i - a_i}{a_i})^2} \times 100 \quad (5.9)$$

where N is the number of samples, and p_i and a_i are the predicted and actual values of the sample, respectively. We want the % relative RMSE to be as low as possible. RMSE is a standard metric in regression which is sensitive to scalability. For example, an RMSE of 1 s in runtime prediction is not acceptable if the actual runtime is 2 s, but can be acceptable if the actual runtime is 1000 s. Expressing the error as a percentage of the actual value solves this issue.

Assessment of Predictors and Estimator: Figure 5.6 shows how best each model can predict the next behavior of running application. Results show that the average errors of

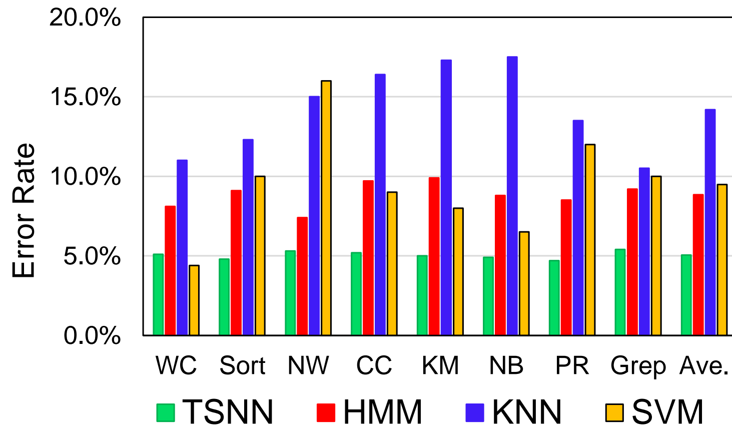

Figure 5.6: Error rate of ML-based predictors

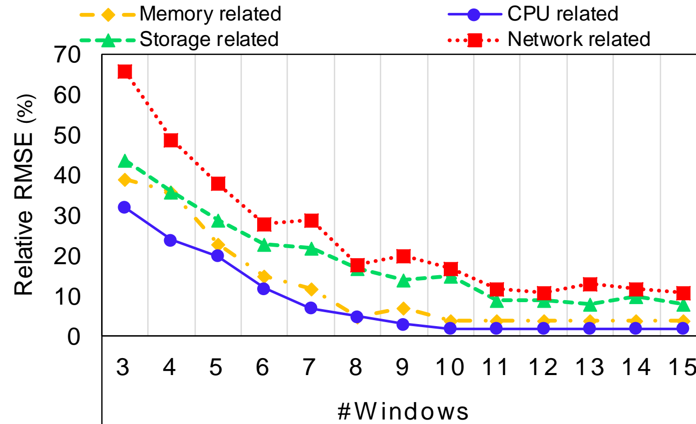

Figure 5.7: Impact of the number of windows on the accuracy of TSNNs

models built by TSNN, HMM, KNN, and SVM are 5.1%, 8.8%, 14.2%, and 9.5%, respectively. Based on these results, we opt to only use TSNN as our predictor whcih is more robust than other models. Figure 5.7 depicts the impact of number of windows (time stamp) on the accuracy of TSNNs. As the result shows, 12 windows data are sufficient to accurately predict the next value for most parameters. The accuracy for CPU related, memory related, storage related, and network related metrics are 98%, 96%, 91%, and 89% respectively. As the Figure presents, ANN reaches to 96.2% accuracy with 780 neurons.

Assessment of optimization algorithms: Figure 5.8 shows the performance of optimization techniques to brute force method. ICA performs faster that any other techniques

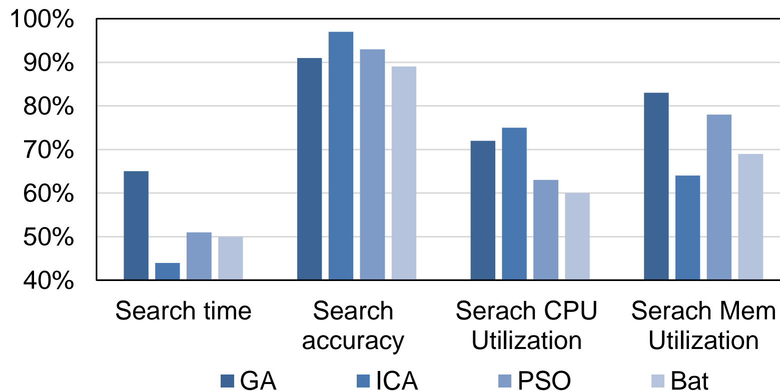

Figure 5.8: Comparison of Optimization techniques (BF as a baseline)

and also has the better accuracy (97%). Moreover, ICA slightly utilizes more CPU but less Memory capacity compared to the other techniques. We observe GA has the worst performance, and PSO and Bat techniques have similar performance. Based on our evaluation, we opt to use ICA as our optimization engine. On average, ICA reaches to its best performance on 65 iteration and we use this number as stop threshold.

Performance and Energy Efficiency

In this section we compare E-Net methodology with other schedulers. To the best of our knowledge there is no off-the-shelf approach for minimizing the EDP of arbitrary IMC workloads on various server types in a heterogeneous cluster. Often the end users provision the most powerful server, but this may lead to excessive costs and energy waste without any significant performance gains. Alternatively, exhaustively profiling the workload on every available server provides accurate EDP estimates, but it is prohibitively expensive in terms of cost, energy, and time. Instead, we chose baseline schedulers described below that can be a good candidate to evaluate and compared with E-Net:

(1) *Oracle scheduler*: This scheduler is an ideal scheduler that has a prior knowledge of the application's behavior and therefor, it provisions the best resources for each phase of an application at runtime.

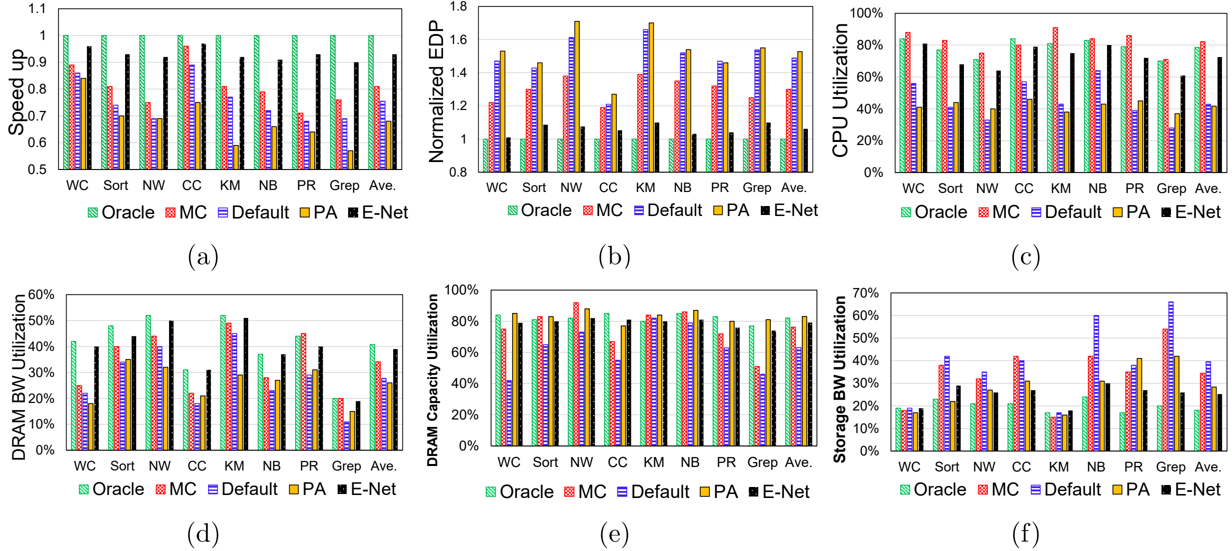

Figure 5.9: Evaluation of different metrics using various schedulers: (a) Normalized speedup; (b) Normalized EDP; (c) CPU utilization across all servers; (d) DRAM bandwidth utilization across all servers; (e) DRAM capacity utilization across all servers; and, (f) Storage bandwidth utilization across all servers.

(2) *Default scheduler*: This is the base scheduler of Spark and Flink. In this case, applications are running on the cluster without any manipulation from outside.

(3) *Matrix Completion (MC) scheduler*: We compare E-Net with a representative matrix completion method or collaborative filtering [62] proposed previously in Quasar [26]. The goal of this scheduler is to optimize the EDP. The advantage of comparing E-Net with this type of scheduler is to understand how E-Net works well compared to a simpler scheme scheduler. For this purpose, we generate the initial dense matrix using our offline benchmarking database. Applications form the rows of the matrix, the server types form the columns, and each cell (j, k) in the matrix contains the value of the EDP for j-th application on the k-th server type. When an application is profiled on two servers, we modify a row with only two values corresponding to those two servers. Using matrix completion, we are able to obtain the predicted the EDP of the application on the rest of the servers.

(4) *Power-Aware (PA) scheduler*: It is a scheduler that allocates the lowest-power platform first.

Performance: Figure 5.9(a) shows the speedup of applications using E-Net and other methods. In this figure, we normalized speedup of each application to the oracle results such that higher value represents a better speedup. We observe that E-Net always outperforms MC scheduler. This is unsurprising as E-Net scheduler is designed to proactively estimate the energy delay product of arbitrary IMC workloads at each phase of the program on various server types at runtime to provision the best energy-efficient configuration in a heterogeneous scale-out environment. On average, E-Net improves the performance by 23% and up to 36% compared to default scheduler. Moreover, E-Net has 14.8% better performance than MC scheduler on average. This is noticeable as this speedup was only achieved by efficient resource allocation, without any software or framework parameter tuning. Results show PA scheduler is not an effective solution for speedup as it performs worse than the default scheduler. Because PA always tries to select the lowest power configuration for resource allocation and hence, its performance is lower than the other schedulers.

Energy Delay Product: Figure 5.9(b) shows the EDP improvement of studied workloads. Again, we normalized the results with regard to the oracle. The lower value is better. Our results show the default scheduler outperforms PA. Because, PA can only allocate low power configuration to the application. As a low power configuration has the lower performance and in the other hand EDP is a metric that favors performance, PA's outcome is a high EDP configuration. Hence, default scheduler works better than PA. Moreover, MC works better than PA. Because MC is more intelligent than PA. MC scheduler is able to learn from the experience of other applications and also the current application's history. Based on the presented results, E-Net improves the EDP by 40% on average and up to 51% compared to default scheduler. Moreover, E-Net reduces EDP 18.3% more than MC scheduler, on average.

In order to compare the benefits of using E-Net over a bare metal cluster where only Spark or Flink are managing the jobs without interfering of any virtualization mechanism such as Xen hypervisor, we performed 15 experiments with a different number of jobs that runs in parallel. Figure 5.10 shows the EDP normalized to the Oracle Scheduler. BM stands

for Bare Metal cluster using default Spark's or Flink's scheduler. The interesting observation is that when the number of jobs is low, BM outperforms all virtualized solutions. However, by increasing the number of jobs, the BM cannot manage the jobs and this is where E-Net's complexity pays off. The results show E-Net co-locates multiple jobs with the same quality of managing single job and still allocates the close to optimal configuration to VMs. In the other hand, the effectiveness of virtualized default scheduler reduces when multiple jobs are running concurrently. Another interesting observation is that by increasing the number of simultaneous jobs, PA starts to outperform virtualized default scheduler. Because by increasing the number of jobs, the ability of default scheduler to properly manage jobs reduces and the performance degrades. Therefore, as the PA can mange jobs with the lower power consumption, it can outperform default scheduler.

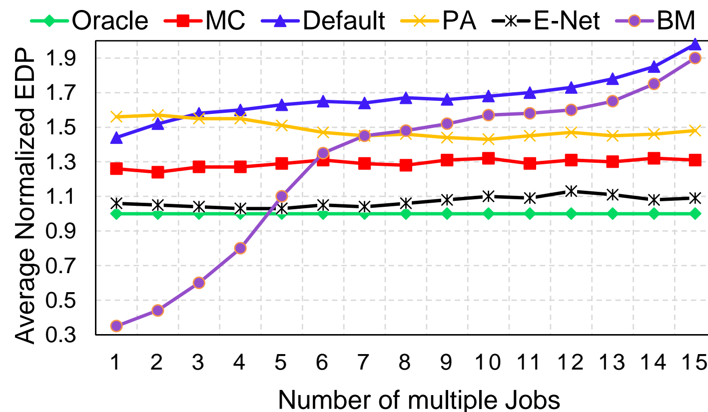

Figure 5.10: Impact of running multiple jobs on EDP

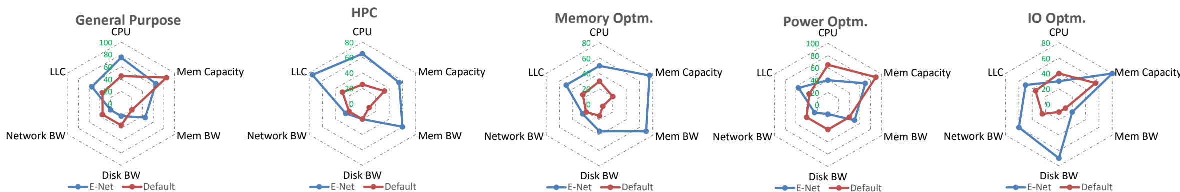

Figure 5.11: Correlation between the server type and average utilization

Utilization: Figure 5.9(c) demonstrates the results for CPU utilization across all servers. As shown, E-Net increases CPU utilization (average across all servers) to 72% versus 43%

with default scheduler which is a 69% improvement. Figures 5.9(d), 5.9(e), and 5.9(f) show the average utilization of DRAM bandwidth, memory capacity, and storage bandwidth during the execution of workloads across all servers. Based on the evaluation results, E-Net increases DRAM bandwidth and memory capacity utilization by 49% and 31% respectively on average compared to default scheduler. However, the results show that the storage bandwidth utilization is reduced by 35% on average. E-Net increases the storage bandwidth by aggregating multiple disks together to alleviate I/O bottleneck and improves the performance. Since the storage bandwidth increases and the usage remains constant, the utilization drops. Hence, this reduction of storage bandwidth utilization is not interpreted as a negative sign.

Figure 5.11 shows the correlation between the server type and utilization. The results clearly show that the default scheduler is not aware of the server types and wastes the resources available on each host. However, E-Net perfectly manages resources in a way that each server has been assigned to a VM that can take advantage of its resources. For example, we can observe that default scheduler is putting too much pressure on Power Optimized servers but it does not use extra resources on memory or IO optimized servers. Whereas, E-Net tries to lower the pressure on power optimized and general purpose servers. At the same time, E-Net can take advantage of HPC and memory optimized server by assigning more VMs to them based on each VM's requirement.

Limitation

E-Net took a first but important step in cluster management to help users provision energy efficient resources for their workloads on a heterogeneous scale-out platform such as the cloud. In this section, we describe the limitations of the current system and possible future directions. At this stage, E-Net is not data size aware. Extending E-Net to consider task-specific features, such as input size, can enable generalization across jobs. In the future, we plan to merge E-Net resource provisioning algorithms in Apache Helix cluster management frame-

work. Extending E-Net for multi-tier services will also require further work. Furthermore, E-Net does not support fault tolerance. However, this will be a straightforward extension if E-Net server is used as a hot-spare mirroring to provide fault-tolerance which requires a continuously replication of all system states between two servers. E-Net does not explicitly consider latency-critical applications or dependencies between application components. It also does not enforce fine-grain priorities between application components or user requests, or optimize for shared data placement. We will address these issues in our future work.

5.6 Conclusion

To address the challenge of resource provisioning for IMC workloads in heterogeneous cloud platforms consist of diverse types of servers, in this work we propose E-Net which is a proactive online resource provisioning methodology. As cloud platforms provide a wide range of server configuration choices, and the applications' performance and power consumption changes at runtime and depends on the chosen configuration, resource provisioning in cloud platforms is a challenging optimization problem with a large search space to navigate. E-Net proactively assigns a suitable hardware configuration to IMC program for energy-efficiency (EDP) optimization at runtime before any significant change occurs in the application's behavior. This helps to save energy without sacrificing performance. E-Net uses time series neural network to predict the next phase of an application. It then uses artificial neural networks to estimate the performance and power consumption of the predicted phase of application on various server configurations. Further, E-Net uses a imperialist competitive algorithm to distinguish close-to-optimal configuration in order to minimize EDP. Compared to Oracle scheduler, E-Net achieves 93% accuracy to allocate the right resource to a given workload at runtime and for each phase of the program. E-Net improves the performance by 23% and the EDP by 40% on average, compared to default scheduler.

Chapter 6

Proactive and ML-based RPS for Data-Intensive Applications

To the best of our knowledge, previous works did not fully address all the following challenges together: proactive behavior prediction of data-intensive applications, selecting a highly accurate machine learning (ML) technique for performance modeling based on real empirical characterization, consideration of VM live migration time overhead, optimizing the performance w.r.t the cloud cost, and considering various optimization techniques to achieve fairness among jobs at the same time.

In response to these challenges, we propose ProMLB-a methodology that proactively predicts the future behavior of running applications by dynamically generating a cross-platform performance model for all the available hardware resources and provisions a near-optimal configuration that maximizes performance per cost, while introducing a low search overhead. The overhead of generating the performance model and finding the optimal configuration is negligible as ProMLB is implemented on a separate server as centralized cluster management, and it does not interfere with on-going application executions.

6.1 Contribution and Novelty of ProMLB

In this work, we propose a practical framework called ProMLB that can address all the aforementioned challenges simultaneously. To this goal, we first analyze various applications' architectural characteristics. Based on this information, a database is built and used for training the prediction models (using time series neural network, K nearest neighbors regression, and hidden Markov model), generating performance models (using multilayer perceptron, and Support Vector Machine), and applying different optimization techniques (Knapsack algorithm, and Cobb Douglas utility function) in terms of performance/cost efficiency and fairness. To be as close as to the real-world cloud providers, we utilize IBM's SoftLayer pricing list to derive a cost model for server platforms in a scale-out environment. The developed cost model takes into account the processor, memory, and disk configurations.

The novelty of this work is outlined in a three-fold manner:

- An Ensemble learning-based proactive phase prediction with high accuracy based on the behavior of the application and underlying execution hardware is devised.
- A non-linear performance model to efficiently estimate the actual behavior of the applications running on different hardware platforms considering the architectural parameters, as well as real-time constraints such as migration time, is deployed.
- A cost-performance trade-off is achieved by employing the Bounded Knapsack algorithm. In order to solve the problem of proactive resource allocation in data centers with minimal processing overheads, a hierarchical approach based decision-maker is employed in the last stage.

The evaluation results show that the phase predictor of ProMLB achieves 92% to predict the future phase change of workloads correctly. On average, ProMLB improves the performance/cost (performance per unit cost) and performance by 2.5x and 42% on an average (up to 70%), respectively. It needs to be noted that this improvement in speedup is

only achieved by efficient resource allocation and without any software or framework tuning overheads. Based on the evaluation results, ProMLB increases CPU utilization efficiency (averaged across all cores), DRAM bandwidth, and memory capacity utilization efficiency by 36%, 53%, and 39%, respectively.

6.2 ProMLB

In this part, we discuss the details of ProMLB and its components.

Overview

Figure 6.1 shows the overview of ProMLB framework. In order to continuously monitor each server's state, a monitoring agent runs on each host. These agents periodically send the host's state, such as architectural information and resource utilization, to the ProMLB server. ProMLB server maintains a database of per-host state and updates it on each interval. ProMLB predicts the future state of application based on the current and previous states. Based on the predicted state and corresponding architectural signature of application, ProMLB generates the application's performance model for all available platforms in the cloud. Afterward, ProMLB solves an optimization problem to find the best platform and configuration that maximize the performance/cost for a specific application at a given budget. Then, ProMLB uses Cobb-Douglas utility function to achieve fair allocation.

ProMLB is designed to maximize the performance per cost of running a data-intensive application on a distributed platform. To achieve this goal, our approach is to use bounded knapsack algorithm. In the Knapsack algorithm, by giving a set of items each with a weight and a value, we must determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. In our problem, we consider each resource as an item and their value is the performance gain that they can add to the system. In our problem, the cost of each resource is equal

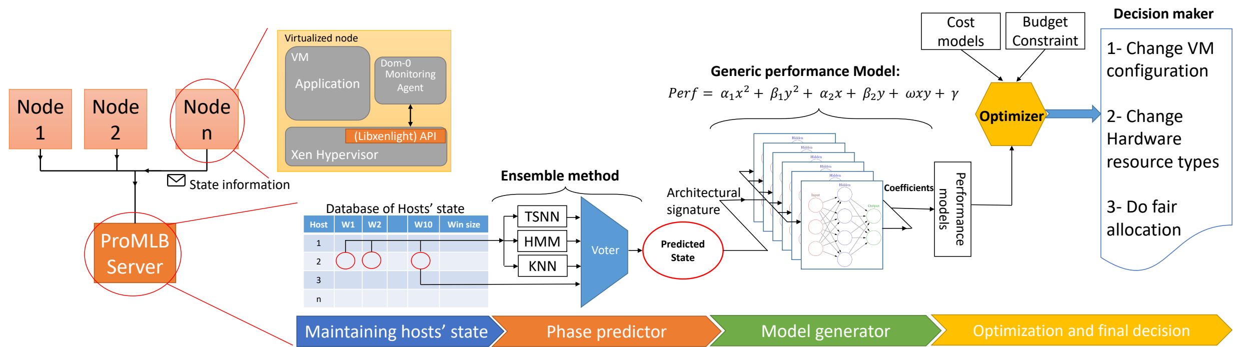

Figure 6.1: ProMLB overview

to the weight of an item in the original problem. The given limit is equal to the total budget of the user to provision resources for its application. We have another restriction that from each item, we must select at least a minimum amount. Because an application cannot start execution unless it gets a core, memory, storage, and network resources. To solve the optimization problem, we need a performance and cost model to determine the performance gain and the cost of adding each resource to the system.

The important point is that as we must assign all resources at the same time, we need a performance model that correlates the performance of adding a resource to all other resources. For example, if we want to add one core to the system, the performance gain of that extra core is dependent to the current resources, and it is not independent from them. Therefore, we need performance models to accurately calculate the performance gain. On the other hand, the cost of adding each resource is independent from other resources. Therefore, the cost models are much simpler than the performance models. ProMLB server consists of four components to deliver all the above functionalities: Phase predictor, performance model generator, optimizer, and decision-maker.

Figure 6.2 shows the block diagram of ProMLB and explains how aforementioned components work together, e.g., how the optimizer and the manager take the knowledge of the predicted results and models in their configuration and allocation decisions.

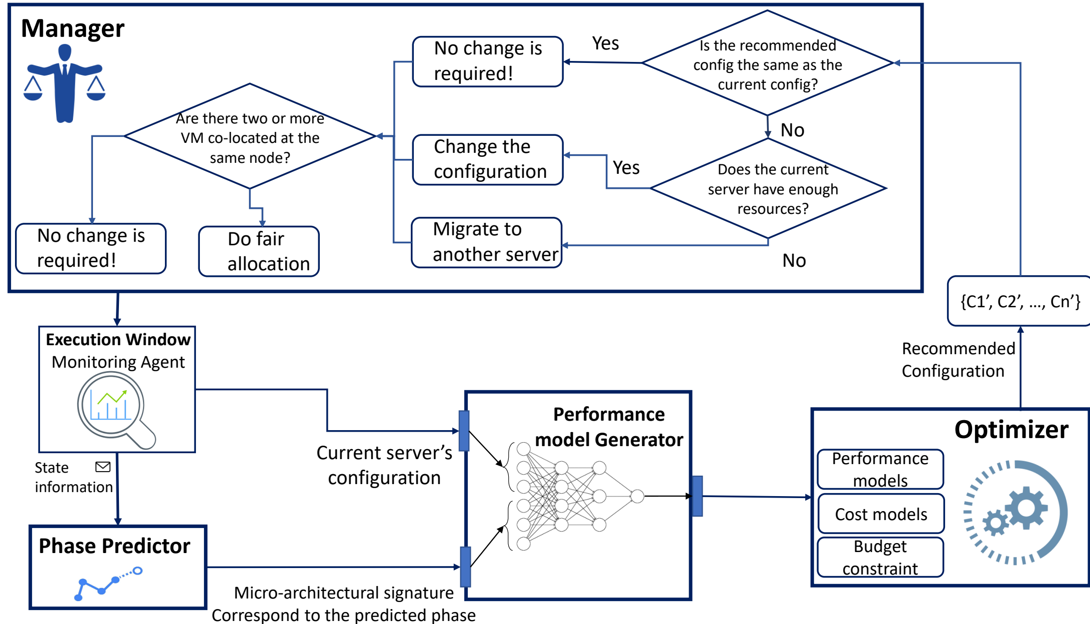

Figure 6.2: Block diagram of ProMLB

Monitoring agent

The monitoring agent is similar that we implemented in section 5.3. It has been implemented in a privileged VM in Xen hypervisor called Dom-0. Alternatively, datacenter operators may decide to host it on the application's VM. The monitoring agent periodically reports the state of the host to ProMLB server. The duration of the period, which is referred to as Window, depends on the application's characteristics. The monitoring agent reports the current state, architectural signature, and the duration of window to the ProMLB server.

Architectural signature

The monitoring agent extracts architectural information of application during each window and reports it to the server. This architectural information is collected through the Intel Performance Counter Monitor tool (PCM) [43] to understand the memory and processor behavior. The information that we use to study the behavior of applications are: available virtual, physical, and shared memory, the cache, buffer space, memory bandwidth utilization, ratio of free to total disk space, storage bandwidth utilization, network Bytes sent and

received, L2 and Last Level Cache (LLC) hits ratio, instruction per cycle (IPC), core C0 state residency, and CPU idle, system, user time. We consider this information as an architectural signature of an application.

In this work, the resource usage pattern is called application's behavior. We showed that the behavior of an application changes if it becomes memory-bound, core-bound, I/O-bound, or idle. Each of those distinct behavior is called a "phase". As an example, when we say an application is in its memory bound phase, it means its memory usage pattern shows the application is memory-intensive. Hence, if the application's behavior changes, it means that its phase is changed too. Therefore, a period of time that application shows a distinct behavior is a phase. We define "window" as follows: a fixed duration of time that monitoring agents periodically sends its state report to the ProMLB server (master node). During each window, an application may have multiple phases. We label the window with a phase that consumes most of the time of that window. For example, if two thirds of a window are memory-bound, we call that window memory-bound. Basically, what our predictors will predict would be the major phase in the next window.

Phase predictor

Similar to E-Net, we equipped ProMLB with a phase predictor to be proactive in order to act before a significant change happens in the behavior of the application and degrades the application's performance. Phase predictor will predict the future behavior of the application based on the current and previous behavior. For this purpose, we employ three techniques, such as time series neural network, hidden Markov model, and K nearest neighborhood. Each of these techniques has its trade-offs. We observed during simulations that accuracy is limited for unseen applications. Therefore, we employ Ensemble learning to boost accuracy. There are several complex neural networks such as CNN and LSTM that provides high accuracy but their complexity is not appropriate for our application. While few works proposed to reduce the complexity and the number of parameters of such networks [83, 85] but still they are

much more sophisticated than our selected techniques. We use the ensemble method, which uses a combination of multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning techniques alone. The accurate prediction is important as we can allocate enough resources to the application before the performance degrades.

Ensemble method

Ensemble learning [100] is a branch of machine learning which is used to improve the accuracy and performance of general ML predictors. We use ensemble learning to enable the use of both eager learning techniques (TSNN, HMM) and lazy learning techniques (KNN), which does not require training. Using the Lazy learning technique enables ProMLB to be more flexible and have better accuracy for unknown applications. In this work, we use Bagging or Bootstrap Aggregation [100], which is an ensemble learning model that is used for predictions. It is a statistical prediction technique where a future state of the application is estimated from voting of prediction results of three models. Each model is exploited to make a prediction, and the results are voted to give a more robust and generalized prediction. If the prediction of all three ML techniques is different from each other, then the voter will select the current state as the final result.

Adaptive performance modeling and cost model generator

In this part, we formulate the performance and cost analysis for different applications in a scale-out environment. The first part of this section is devoted to performance modeling. The second part is to formulate the dependency of the price that a subscriber must pay for utilizing different server configurations. This cost model is based on what we presented in section 4.4. We then present the developed models to formulate the performance improvement of each application with respect to the baseline hardware configuration. These models will be exploited by the optimizer in the next step to select the most performance- and

cost-efficient server configuration for a given application.

Adaptive performance modeling

One of the novel contributions of this paper is to adaptively generate a performance model for each phase of applications dynamically. This leads to a more accurate model and helps the optimizer to select the best configuration. Figure 6.3 is an example to illustrate that each application has a different performance model depending on its phase and the server platform. Offline analysis of our applications shows that the performance of data-intensive applications is a convex function of servers' parameters such as core count and core's frequency. Based on the analysis of our characterization, a generic performance model can be developed. However, this generic model has to be adopted for each application. As a panacea to automatically tune the generic model depending on the architectural signature of the application, we employ Artificial Neural Network (ANN) here [35]. ANNs are a class of machine learning technique that maps a set of input parameters to a set of target values.

We formulate each server's performance as the product of per-processor performance and the number of processors in each server. Regarding servers' configuration, the parameters that can be configured are core count, core frequency, DRAM bandwidth and capacity, storage bandwidth. Therefore, there are nine different performance models from the combination of those parameters. As the performance does not scale linearly with the parameters, such as the number of cores, a nonlinear modeling is required. Following Equation demonstrates the generic model:

$$Perf = \alpha_1 x^2 + \beta_1 y^2 + \alpha_2 x + \beta_2 y + \omega xy + \gamma$$

Where x, y $\in \{core, freq, DRAMBW, DRAMcap, StorageBW\}$ and $x \neq y$.

In order to capture this non-linearity effectively, we chose a Support Vector Machine (SVM) [109] to fit the performance models. SVM analysis is a popular machine learning tool for regression. Based on our database, we fit these models using SVM in order to find

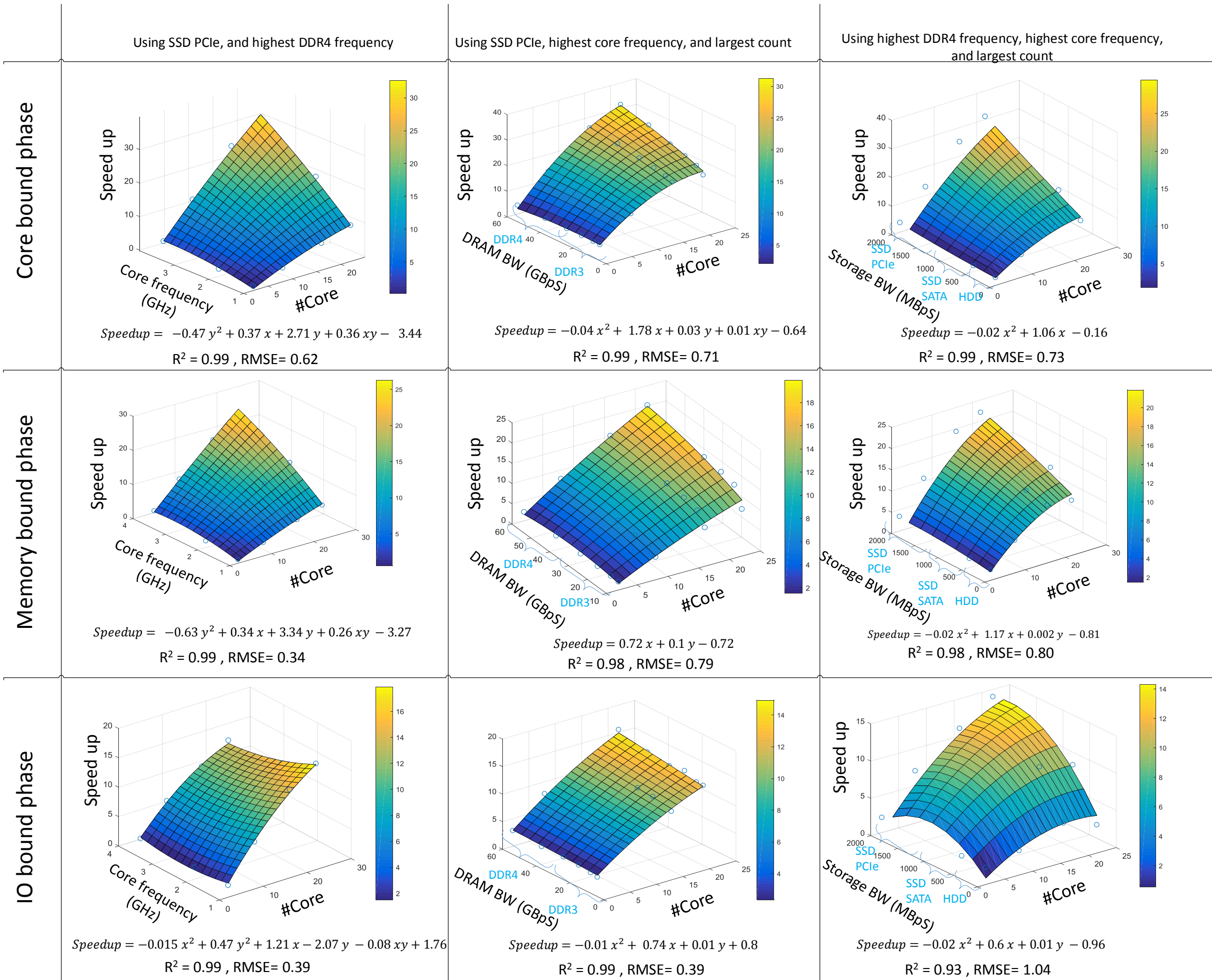

Figure 6.3: Performance model generated for workload graph analytic from Flink framework

the coefficients. Once the coefficients are calculated, we use them for training the ANNs in order to map the architectural signature of applications to those coefficients.

Nine three-layer fully connected ANNs were trained by our training database to adopt the generic performance model for each application based on the architectural signature. We started with a simple three fully connected layer neural network. We found out this model achieves our desired accuracy. Therefore, we did not use a more complex model, such as a convolutional neural network. Each ANN has 17 inputs, 230 hidden neurons, and six outputs. We used Grid Search to find the best number of hidden neurons. Inputs of neural networks are the architectural signature. Each output neuron stands for a coefficient. ANNs generate

the performance models in parallel. In the next section, these models will be used to find the best platform and configuration. Figure 6.3 shows a subset of generated performance models for three different phases of graph analytic application in the Flink framework. In each sub-figure, X represents the number of cores, and Y stands for the other parameter.

The advantage of this approach is that we can accurately model the performance of applications at each phase of their execution for various type of servers and improve the server selection. An appropriate resource provisioning will decrease the execution time of subscriber's job, increases the resource utilization of scale-out infrastructure and eventually brings economic benefits for both subscriber and provider. This is important because performance improvement in datacenters translates into millions of dollars revenue per year for cloud provider and also it decreases the cost for subscriber and make cloud services more attractive for the end users.

Cost Modeling

This cost model is exactly what we presented in section 4.4 in MeNa with this change that data is collected in January 2019. All cost equations are the predicted charge that subscribers must pay in dollar for renting a bare metal server (on a monthly basis) on the IBM SoftLayer, which includes the power, cooling, and maintenance related costs of the server.

Optimizer

For a given application and workload, our goal is to find the optimal or a near-optimal server configuration that simultaneously satisfies the performance requirements with minimal operational cost. For this purpose, we use the Bounded Knapsack algorithm to solve the aforementioned optimization problem.

Bounded Knapsack solution

This solution was introduced in MeNa [69] to select the best memory configuration to maximize the performance/cost. We discussed it in detain in section 4.5. In this work, we use the Bounded Knapsack solution to select the optimal server configuration (not only memory). The result of this optimization is the recommended configuration to the manager. The budget is a constraint that the user must provide. The result of solving the optimization problem is a set of configuration such as the number of sockets, number of nodes, the number of cores, core frequency, memory capacity, memory bandwidth, storage capacity, and storage bandwidth. The optimizer recommends this configuration to the manager. It is the responsibility of the manager to decide about the action which is needed to take for scaling the current platform to make it as close as to the recommended configuration for the targeted VM.

Manager

After finding the optimal configuration, the manager takes actions to allocate or adjust the resources assigned to the applications. The manager of ProMLB is working in the same way as the E-Net decision maker discussed in section 5.3. Actions that can be executed by the manger are as follow:

The first action can be Dynamic voltage and frequency scaling (DVFS) which is the adjustment of voltage and speed settings to increase or decrease CPU frequency. If it is required, the manager can increase or decrease the number of CPU cores assigned to the application (hot-(un)plugging of resources such as memory and cores). Moreover, the manager can change VM configuration and add allocated storage or remove them. It is also feasible to increase or decrease memory capacity. The last action will be to migrate VM to a different node. Live migration is performed by an underlying mechanism (Xen Hypervisor). The manager migrates a VM when the migration latency is predicted less than half of the window's time and also when there are not enough resources on the current server.

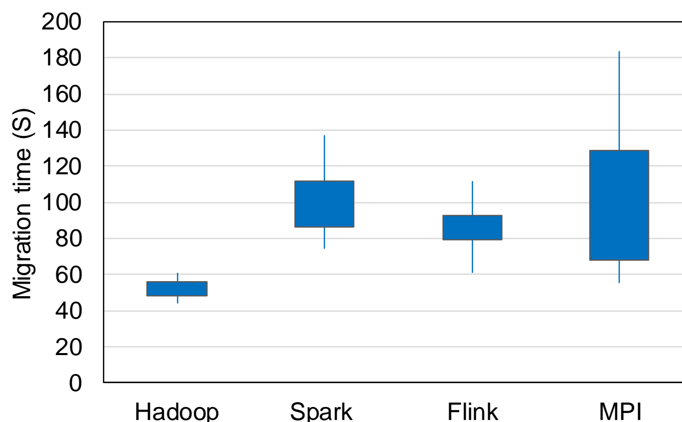

Figure 6.4: Migration time

Predicting migration time

We use the same model of migration modeling discussed in section 5.3. More details are available in [82].

Figure 6.4 shows the variation of migration time for different data-intensive frameworks. In this study, we set the window size equal to three times (3X) of migration time because it gives better prediction accuracy (Figure 6.6 shows the impact of window size on accuracy).

Fair allocation:

Another aspect to consider is fair allocation. When running multiple VM on a node, the manager uses Resource Elasticity Fairness (REF) [132] in order to allocate the resources among VMs, as co-scheduling multiple VMs on a single server could result in interference. REF is a fair allocation mechanism that satisfies three game-theoretic properties (sharing incentives (SI), envy-freeness (EF), and Pareto efficiency (PE)) using Cobb-Douglass utility function. In this work, we begin with the space of possible allocations. We then add constraints to identify allocations with the desired properties as follow:

Suppose multiple VMs share a server with R types of hardware resources. Let $x_i = \{x_{i1}$, ..., $x_{iR}\}$ denote i-th VM's hardware allocation. Further, let $u_i(x_i)$ denote i-th VM's utility.

Following equation defines utility within the Cobb-Douglas preference domain:

$u_i(x_i)= a_{i0} \sqcap_{r=1}^{R} x_{ir}{}^{a_{ir}}$

The parameters a_i=$\{a_{i1},...,a_{iR}\}$ quantify the elasticity with which a VM demands a resource. Let C_r denote the total capacity of resource r in the system. We can find a fair multi-resource allocations given Cobb-Douglas preferences with the following feasibility problem for N virtual machines and R resources:

Find x subject to:

1) $u_i(x_i) \geq u_i(x_j)$ i,j∈[1,N]

2) $\frac{a_{ir}}{a_{is}} \frac{x_{is}}{x_{ir}} = \frac{a_{jr}}{a_{js}} \frac{x_{js}}{x_{jr}}$ i,j∈[1,N]; r,s∈[1,R]

3) $u_i(x_i) \geq u_i(C/N)$ i∈[1,N]

4) $\sum_{i=1}^{N} x_{ir} \leq C_r$ r∈[1,R]

Where $C/N=\{C_1/N,...,C_R/N\}$. In this formulation, the four constraints enforce EF, PE, SI, and capacity. The outcome of applying Cobb-Douglass utility function is a fair resource allocation among multiple VMs running on a server.

Resource isolation:

In order to decrease the side effects of resource contention and interference, we enforce resource partitioning and isolation techniques. We employ core isolation (thread pinning to physical cores), to constrain interference context switching. We employ the Cache Allocation Technology (CAT) available in Intel chips [49] to isolate last level cache (LLC). The size of cache partitions can be changed at runtime by reprogramming MSR registers. We also use the outbound network bandwidth partitioning capabilities of Linux's traffic control. We employ the `qdisc` [77] to enforce bandwidth limits. To perform DRAM bandwidth partitioning, the manager monitors the DRAM bandwidth usage of each application using Intel PCM to co-locate jobs on the same machine where it can accommodate their aggregate peak memory bandwidth usage.

6.3 Implementation

In this section, we present our experimental system configurations and the setup. We first introduce the frameworks and the workloads we consider for evaluating the ProMLB. We then describe our hardware platform which runs the ProMLB server.

Workloads

In our study, we used Hadoop MapReduce version 2.7.1, Spark version 2.1.0 in conjunction with Scala 2.11, Flink version 1.3.3, and MPICH2 version 3.2 installed on Linux Ubuntu 16.04 LTS.

For a building and training of ProMLB, we target various domains of data-intensive workloads such that of microkernels, graph analytics, machine learning, E-commerce, social networks, search engines, and multimedia, totally 19 workloads. The size of input is in the range of 10 GB and 2 TB. We use BigDataBench [119] and HiBench [46] for the choice of big-data benchmarking. The selected workloads have different characteristics such as high-level data graph and different input/output ratios. Some of them have unstructured data types and some others are graph based. Also, these workloads are popular in research and are widely used for demonstration of techniques. For validation of ProMLB, we used CloudSuite [29] workloads: Data Analytics, web search, Graph Analytics, and In-memory Analytics.

Figure 6.5 clarifies how we divided our workloads and dataset. First part is devoted for developing the system, and the second part is devoted for the evaluation of our entire system. During the development part, we used 19 workloads from two suites (BigDataBench and HiBench) to train our models. To evaluate our models during this part, we partitioned our dataset to two sets (unseen dataset for testing, and seen dataset for the training and validation). In this part, data from all 19 workloads are aggregated together and we randomly leave out 20% of the data for the unseen data points. Then we applied 5-fold cross-validation technique on the 80% remaining data. The common schemes of cross-validation are m-fold

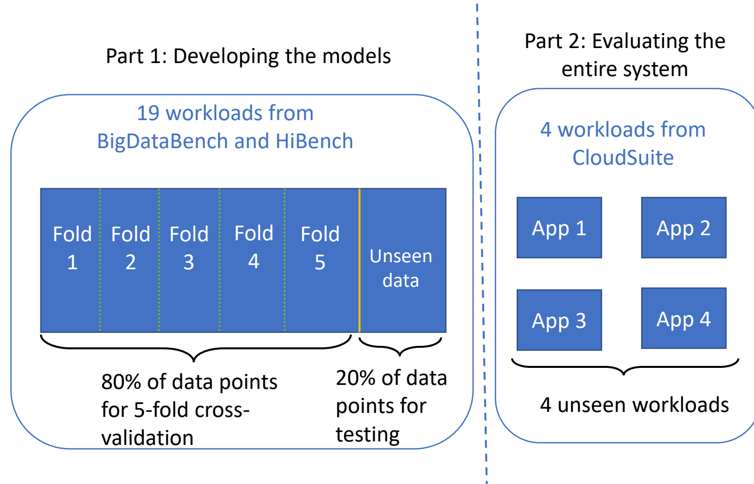

Figure 6.5: Workloads and the dataset division for the training and testing phases of ProMLB

cross-validation. In m-fold cross-validation, the dataset is randomly divided into m subsets or folds and repeated m times. Each time, one fold is reserved as a test dataset to validate the model and the remaining m-1 folds are used for training of the model. Then, the classification accuracy across all m trails is computed. Figure 6.8 is related to this part of our experiments. As you can see, the training subfigure is related to the accuracy of training folds (m-1 folds) and validation subfigure is related to the validation fold (the remained fold). Then the testing subfigure is related to those 20% data that we leaved out from dataset as unseen data points. After building the models and in the second part, to evaluate ProMLB with a completely new unseen workloads, we selected 4 workloads from CloudSuite which we never used them to train or build our models. Results presented in figure 6.9(b) are related to this part of our dataset.

Hardware Platform

We tested ProMLB on our 40-node cluster. Our cluster includes servers of twelve different configurations shown in Table 5.1.

ProMLB Prototype

We implemented the ProMLB prototype as a Java application running on Linux Ubuntu 16.04 LTS. ProMLB is merged into Apache CloudStack [106]. At the current stage, ProMLB can not be used for container-based systems such as Kubernetes. We will address these issues in our future works. ProMLB currently can manage Hadoop, Spark, Flink, and MPI based data-intensive applications. At this stage of implementation, ProMLB does not support fault tolerance. This will be a straight forward extension if ProMLB server is used as a hot-spare mirroring to provide fault-tolerance which requires a continuously replication of all system states between two servers. ProMLB does not explicitly consider latency-critical applications or dependencies between application components. It also does not enforce fine-grain priorities between application components or user requests, or optimize for shared data placement.

6.4 Evaluation of ProMLB

In this section, first we report the experimental results in terms of overhead and accuracy. We then compare ProMLB scheduler with other schedulers.

Overhead

In this subsection, we report the overhead of ProMLB, including the time used to collect training data, training the performance models, and searching for optimum configurations. The collecting data has the highest overhead, 8.3 hours on average and up to 10.2 hours for each workload (using 1 GB input data). Model training of each predictor took 31 minutes

Table 6.1: Average prediction time of each predictor.

	Average prediction time
KNN	74 ms
HMM	351 ms
TSNN	580 ms
Ensemble	22 ms

on average and training of ANNs took 73 minutes using two Nvidia GeForce RTX 2080 on a 16 core processor. It should be noted that training time is a one-time cost and it is even an offline cost. It only is required when we are building the system. When the development of system is finished and the system is under the deployment, we do not have such overheads. Compared to the manual configuration this overhead is still attractive as the target of ProMLB is the big data applications that repeatedly run in data centers for months or even longer. In this usage scenario, this one-time cost is amortized with a very large number of runs. Therefore, the additional overhead per run is very low. Table 6.1 presents the average prediction time of predictors. It should be noted that the average of prediction time is around 607 ms.

ProMLB does not need to redo the process of data collection when a new application or a server comes. In this case, when the predictor encounters a new behavior, it saves the trace and reports it to the manager. New traces will be added to the database to retrain and update the models offline. When the new model is ready, it can easily be replaced with the old one without any significant interrupt in the execution of the manager. Searching optimal configuration is done very fast, in seconds. This time is negligible compared to the window's size (few minutes). Moreover, this time will not be added to the execution time of workloads as this computation is being performed on ProMLB server while workloads are performing their normal execution without interrupt.

Corresponding to the resource utilization overhead of ProMLB, we should mention that all components of ProMLB such as predictor, performance model generator, optimizer, and manager are running on the ProMLB server (on the master node). The only component

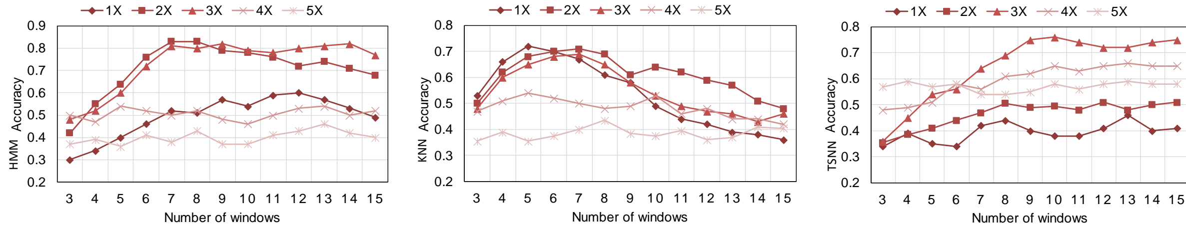

Figure 6.6: Impact of window's size and number of windows on accuracy of predictors

running on the worker nodes is monitoring agent. Therefore, when the ProMLB is managing the cluster, the overhead of total resource utilization of cluster is 1/N in which N is the number of nodes in the cluster. Because only one node is devoted to the ProMLB and the rest of the nodes are running the jobs. Hence, regardless of the memory usage of predictors or the CPU utilization of optimizer, these components are running on master node and they do not have any interfere with the worker nodes.

Accuracy

In this work, we used 20% of our data for testing the models as an unseen data. Rest of the data was used for training and validation through the five fold cross-validation.

Figure 6.6 demonstrates how the window size affects the accuracy of ML techniques. In this work, each timestamp entry of time series neural network is a phase of application. The number of delay for our time series network is 10 windows. We use 8 windows information to train the HMM. We then use this information to predict the next phase. After each window, the HMM model will be retrained and therefore, training of our HMM is online. In this study, we set k = 7 as equal to the number observing windows for KNN.

Each predictor is basically designed to predict what would be the major phase in the next window. For example, if it predicts the distribution of phases in the next window is as follow: 40% of the window time memory bound, 25% of the time core bound, 20% of time I/O bound and the rest is idle, then the predictor concludes that the next window is memory bound which means the application is memory-intensive for the most of the time

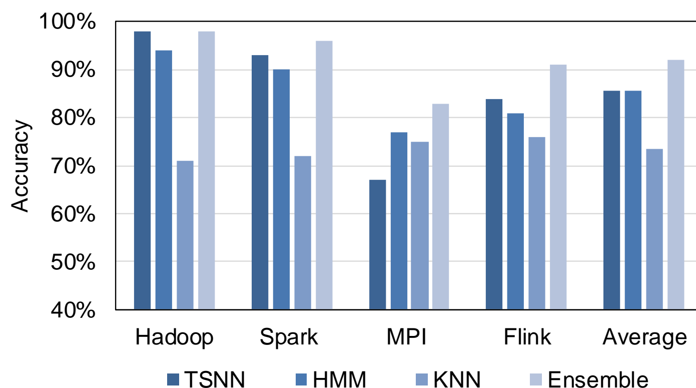

Figure 6.7: Overall accuracy of predictors

in the next window of execution. The accuracy of phase prediction is calculated as follow: Number of windows that predictor correctly predicted the phase / total number of windows. For example, if the prediction for 8 windows from a total of 10 windows is correct, then the accuracy is 80%. The results presented in Figure 6.7 are based on this type of calculation for accuracy.

Figure 6.7 summarizes a validation of the accuracy of the phase predictor engine in ProMLB. The result shows that the accuracy of ensemble method is much higher compared to each ML technique. Ensemble method achieves 92% accuracy to correctly predict the next phase of workloads. It is interesting to observe that the behavior of Hadoop, Spark, and Flink frameworks are more predictable, compared to MPI based applications. it should be noted that that without enforcing any isolation technique, the accuracy will drop to 78%. Therefore, it is important to avoid of interference between co-located VMs for better resource allocation.

Moreover, we also wanted to show that how much the major time prediction is accurate for each window. For example, suppose that the predictor predicts that the major phase in next window would be memory bound for 60% of the time. After the execution of application in that window If we find out that the application was memory bound for 65% of the time, although the prediction was true, but there is 5% difference in the prediction. The results

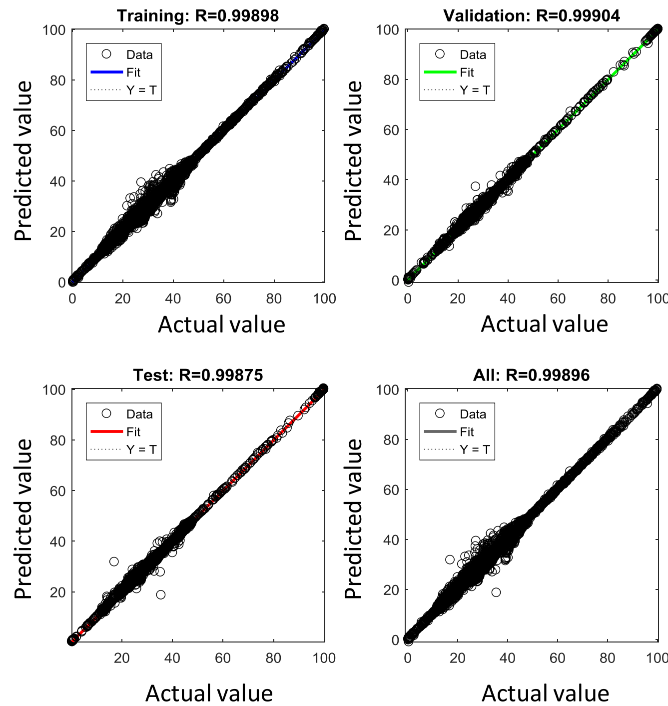

Figure 6.8: Scatter plots of prediction values versus real measurements for 25920 data points

presented in Figure 6.8 shows the actual value and the predicted value in a scatter view by our predictors. R value clearly shows that the models are fairly accurate across the entire configuration space: all data points are located around the corresponding bisectore, indicating that the predictions and estimations are close to the real measurements. Overall, we find that there are not many outliers in our models, indicating that they can be used for optimizing the performance.

Performance and Cost Efficiency

To have a comprehensive comparison between ProMLB and other resource provisioners, we consider the following systems:

(1) *Oracle*: This is an ideal system that has a full prior knowledge of application behavior and therefor it allocates the optimal resources at runtime.

(2) *Default*: This is the default system without any manipulation from outside.

(3) *Matrix Completion (MC)*: Matrix completion method or collaborative filtering [62] proposed in Quasar [26].

(4) *Cherry-Pick scheduler*: One of the closest to ours is CherryPick which uses regression model for performance estimation and Bayesian optimization to find the right configuration.

(5)*Ernest scheduler*: Ernest [112] predicts the runtime of distributed analytics jobs as a function of cluster size and provisions a cost optimal configuration. However, Ernest cannot infer the performance of new workloads on a VM type without first running the workload on that VM type and also is not a dynamic approach.

Performance:

Figure 6.9(a) shows the speed up of ProMLB over the baseline for CloudSuite workloads. On average, ProMLB improves the performance by 42% and up to 70%. This speed up was only achieved by efficient resource allocation, without any change in the applications or frameworks. ProMLB is performing 14% better than the state of the arts approaches because it is proactive and dynamic as well. We observe that MC outperforms Ernest and CherryPick techniques as MC is able to dynamically change the configuration. Figure 6.9(b) shows the performance/cost improvement of studied workloads. On average, ProMLB can improve the performance/cost by 2.5X compared to default scheduler. The interesting observation is that Ernest performs better than MC and CHerryPick in terms of Performance/Cost. The reason is that Ernes is a cost aware approach but ProMLB is still performing 15.6% better than Ernest.

Utilization:

Based on results presented in Figure 6.9(c), ProMLB increases CPU utilization (average across all cores) to 67% versus 49% with baseline which is a 36% improvement. Figures 6.9(d), 6.9(e), and 6.9(f) show the utilization of DRAM bandwidth, memory capacity, and

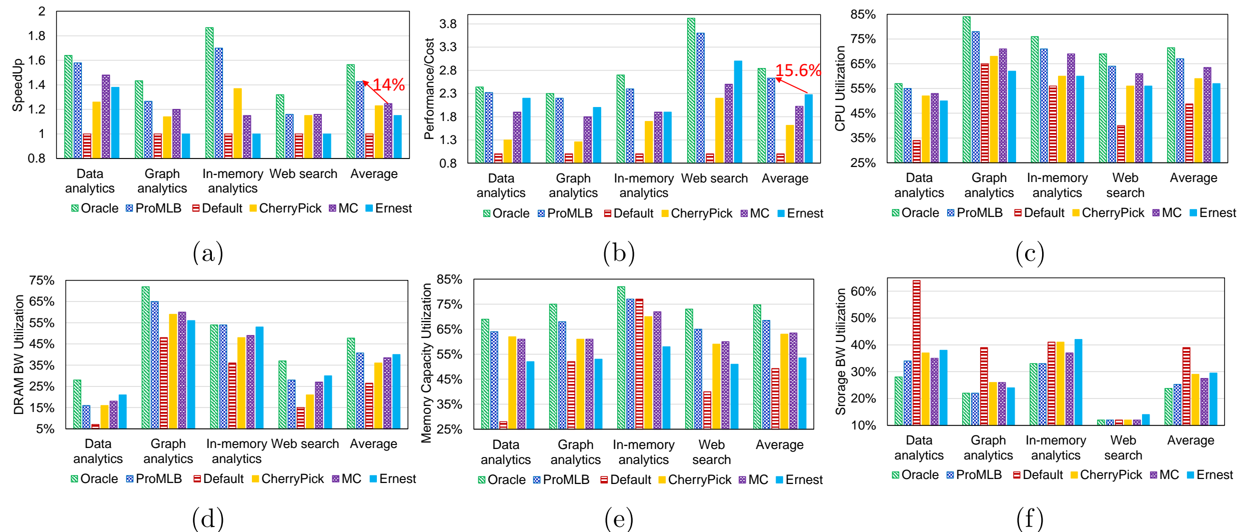

Figure 6.9: Evaluation of different metrics using various schedulers: (a) Normalized speedup; (b) Normalized Perf/Cost; (c) CPU utilization across all servers; (d) DRAM bandwidth utilization across all servers; (e) DRAM capacity utilization across all servers; and, (f) Storage bandwidth utilization across all servers.

storage bandwidth during the execution of workloads. Based on the evaluation results, ProMLB increases DRAM bandwidth and memory capacity utilization by 53% and 39% respectively, on average. Our results indicate that the storage bandwidth utilization was dropped by 35% on average. ProMLB increases the available storage bandwidth to the applications by aggregating multiple disks together. By increasing the storage bandwidth and keep remaining the number of disk accesses, the disk utilization decreases and this degradation does not have a negative impact on the performance.

6.5 Conclusion

In this work, we propose a proactive online resource provisioning methodology called ProMLB to address the challenge of resource allocation for data-intensive workloads in scale-out platforms. A wide range of server configuration choices are available in the cloud, and it creates a large search space to navigate for selecting an optimal configuration. Moreover, the ap-

plications' performance depends on the chosen configuration, and it makes the optimization problem even harder. In this work, to reduce the complexity, the application's behavior is first characterized into a core, I/O, or memory bound. Then, the ensemble learning based prediction is employed to predict the next phase of the application.

Further, the performance models for predicted application behavior across different platforms is derived. As the cost for chosen configuration plays a key role in resource allocation, ProMLB uses an optimization technique to distinguish a close-to-optimal configuration in order to maximize performance per cost. Compared to the oracle scheduler, ProMLB achieves 91% accuracy to allocate the right resource to workloads. ProMLB improves the performance and resource utilization by 42.6% and 41.1%, respectively, compared to baseline scheduler, on average. Moreover, ProMLB improves the performance per cost by 2.5X on average.

Chapter 7

Railroading of Resource Sharing-based Attacks on the Cloud

The heterogeneity of resources and the diversity of applications on the cloud motivated the need for resource provisioning systems (RPSs) to meet the users' performance requirements while maximizing the resource utilization to achieve cost-efficiency. On the other hand, resource sharing-based attacks, such as side-channel, transient execution, rowhammer, and denial of service attacks, exploit shared resources to leak sensitive data or hurt the performance of a victim. Although mounting resource sharing-based attacks on the cloud is trivial once the attacker virtual machine (VM) is co-located with the victim VM, the co-location requirement with the victim limit the practicality of resource sharing-based attacks on the cloud. In this chapter, we show that RPSs can be exploited to solve the co-location challenge of resource sharing-based attacks in the cloud. In particular, we propose a new attack, called *Cloak & Co-locate*, which utilize adversarial evasion attacks to force RPSs to co-locate attackers' VMs with targeted victims' VMs. Specifically, *Cloak* is a fake trace generator (FTG) that is wrapped around an adversary kernel in order to force RPSs to *Co-locate* it with a specific victim's VM, while also evading from detection and migration by the RPS.

7.1 Background and Threat Model

The performance unpredictability problems that stem from platform heterogeneity, resource interference, software bugs and load variation [102], are well-studied in the public cloud. However, there are other challenges that lead to security threats. Below we discuss related vulnerabilities with respect to the resource provisioning systems in the cloud. In particular, we show that RPSs hides security vulnerabilities, since they enable an adversary to extract information about an application's type and the infrastructure's characteristics.

Background

As different clients share hardware resources in the public cloud, isolation becomes a core security challenge for the cloud computing paradigm, which consequently enables adversaries to exploit it for their malicious desires.

A distributed attack [10] aims to retrieve sensitive information, or degrade the performance of a number of computing resources on a distributed system, where each computing resource performs processing of a part of the overall system activity. The retrieved information may, for instance, be a set of encryption keys that can be exploited to compromise the functionality of the whole distributed system. A distributed attack may also be used to retrieve information about the cloud infrastructure. In the following, we describe some characteristics and provide more details on such attacks.

Definition 1 A distributed attack over a set M_{vic} of virtualized instances running in a distributed system S, is defined as the tuple $DSCA = (S, M_{vic}, D, M_{mal}, A, CP, EP)$ where: S is a distributed system; M_{vic} are the VMs that are targeted by the attack; D is the distributed dataset to be compromised (partially or totally); M_{mal} are malicious VMs, co-located with the victim VMs; A is a set of local attack techniques (such as side channel, denial of service, or resource freeing attack); CP is a protocol to coordinate the attacker VMs in M_{mal}; EP is a protocol to exfiltrate data.

We consider $D = \{d_1, \ldots, d_n\}$ a dataset to be processed by the distributed system S

$= \{s_1, \ldots, s_n\}$ implemented on a set of VMs $M_{vic} = \{m_{vic1}, \ldots, m_{vicn}\}$ on a virtualized platform. Each component s_i of S processes data d_i locally and runs on its own VM, m_{vici}. To perform the distributed attack, the adversary sets up a number of malicious VMs at least equal to the number of M_{vic}) $M_{mal} = m_{mal1}, \ldots, m_{maln}$, co-located with the victim instance M_{vic}. The adversary also masters a set $A = a_1, \ldots, a_m$ of local attack techniques, i.e., Flush+Reload.

The objective of a distributed attack is to first attack each component of the system s_i running on m_{vici} through m_{mali} running local attack technique a_j to retrieve d_i. The synchronization between attack instances and a central server may be performed using a coordination protocol CP. A protocol EP may be used to control attacking instances remotely, and to send collected information to a remote server to exfiltrate sensitive data. In the following, we briefly explain three well-known local attacks on a distributed system:

Side Channel Attack (SC)

Lack of enforced isolation can create side-channels, due to the sharing of physical resources like processor caches, or by mechanisms implemented in the virtualization layer. A side-channel is a hidden information channel that differs from traditional "main" channels (e.g., network) in that security violations may not be prevented by placing protection mechanisms directly around data. A side-channel attack exploits a side-channel to obtain important information. SC attacks may be classified according to the type of exploited channel. Timing attacks and cache-based attacks are two main classes of SC attacks, where the processor cache memory is often exploited by adversaries to obtain information. There are attacks that attempt to extract confidential information from co-scheduled applications, such as private keys [139]. Zhang et al. [137] proposed a system that launches side-channel attacks in a virtualized environment. Wu et al. [122] used the memory bus of an x86 processor to launch a covert channel attack and degrade the victim's performance. This type of attack requires the adversarial VM to be co-located with the victim VM.

Denial of Service Attack (DoS)

Denial of service attacks hurt the performance of a victim service by overloading its resources. In cloud settings specifically, they can be categorized into two types; external and internal (or host-based) attacks. Internal DoS attacks take advantage of IaaS cloud multi-tenancy to launch adversarial programs on the same host as the victim and degrade its performance. For example, Ristenpart et al. [95] showed how an adversarial user could leverage the IP naming conventions of IaaS clouds to locate a victim VM and degrade its performance. Cloud providers are starting to build defenses against such attacks by increasing the instances of a service under heavy resource usage. This means that DoS attacks that overload a physical host are ineffective. However, Bolt [22] showed they can perform DoS attacks in a way to make them resilient against such defenses by avoiding resource saturation.

Resource Freeing Attack (RFA)

Resource-freeing attacks also hurt a victim's performance, while additionally forcing it to yield its resources to the adversary [110]. While RFAs are effective, they require significant compute and network resources, and are prone to defenses, such as live VM migration. Delimitrou and Kozyrakis [22] showed that it is possible to launch host-based attacks on the same machine as the victim that take advantage of the victim's resource sensitivity, and keep resource utilization moderate, thus, evading defense mechanisms.

Threat Model

In this chapter, we target Infrastructure as a service (IaaS) providers that operate public clouds for mutually untrusting users. Multiple VMs can be co-located on the same server. Each VM has no control over where it is placed, and no priori information on other VMs on the same physical host. For now, we assume that the resource provisioning system is neutral with respect to detection by adversarial VMs, i.e., it does not assist such attacks or employ additional resource isolation techniques to hinder attacks by adversarial users. Moreover,

RPS is considered ideal and treats all the VMs in a fair manner and places them according to workload characteristics rather than its place of origin or intention. Moreover, we assume black-box access to the RPSs, were we do not know the underlying model that the RPS is using for placing VM instances. Thus, we can only create VM instances and observe the placing outcome.

In this paper, we assume two types of VM as follow:

- **Adversarial VM:** An adversarial VM has the goal of getting co-located with victims and evade from detection mechanism embedded into resource provisioning system to negatively impact victims' performance or steal information.
- **Friendly VM:** This is a benign VM scheduled on a physical host that runs one or more applications. They do not employ any schemes, such as memory pattern obfuscation, to prevent detection by an adversary.

In this chapter, we consider the worst case for an attacker in which if the adversarial application does not change its behavior and architectural signature, the RPS will detect it. The detection mechanism can be any arbitrary technique. Therefore, adversarial applications are required to change its microarchitectural and system level profiling trace to prevent the detection. It should be noted that changing the microarchitectural and system level profiling trace does not mean hacking the performance counters or accessing the RPS's database, but it means that the adversary application changes its behavior, e.g., using CPU more, or performing dummy memory access to increase the cache misses or memory bandwidth usage. The direct result of such behavior is a change in the state of the system, without hacking the system. Additionally, the adversary must remain co-located with the victim. In the case of a change in the behavior, there is a high chance that RPS migrates the adversary VM to another host. For example, Figure 5.1 demonstrates how an applications' behavior changes during the runtime. This is the another challenge of attack and we are going to address it in this chapter.

7.2 Cloak & Co-locate

Contention-based attack's setting

Adversaries are rarely interested in a random service running on a public cloud. They need to pinpoint where the target resides in a practical manner to be able to perform DoS, RFA, or SC attacks. This requires a launch strategy for co-location and a mechanism for co-residency detection. The attack is practical if the target is located with high accuracy, in reasonable time and with modest resource costs. Performing any distributed attack requires a number of prerequisites steps detailed in the following:

Finding physical hosts running victim instances

To perform any attacks based on co-location, the attacker needs several VM launching strategies to achieve co-residency with the victim instance, which is impractical and not feasible. A pre-condition for the attack is that the malicious VMs reside on the same physical host as victim VMs, as side-channel and RFA attacks are performed locally. The first and main step is thus to find the location of physical hosts running victim VMs. Several placement variables such as datacenter region, time interval, and instance type are important to achieve co-residency. These variables may be different among IaaS clouds. However, the application type is considered as an effective factor in the placement strategy [136]. Let $P(m_{mali})$ be the probability of instance m_{mali} to be co-resident with instance victim m_{vici}. The value of P will be raised by increasing the number of launched attack instances. To make sure that both attacker and victim VMs achieve coresident placement, the adversary can perform co-residency detection techniques such as network probing [48]. The attacker can also use data mining techniques to detect the type and characteristics of a running application in the victim VM by analyzing interferences introduced in the different resources to increase the accuracy of co-residency detection [22].

Evasion from detection and migration

There are various techniques to detect an attack in virtualized environment. For example, as side-channel attacks are very fine-grained attacks, the detection of such attacks requires high-resolution information, mainly provided by Hardware Performance Counters (HPCs) [81, 97]. The HPCs are a set of special-purpose registers built into modern microprocessors to capture the trace of hardware-related events such as last-level cache (LLC) load misses, branch instructions, branch misses, and executed instructions while executing an application. Those events are basically used to profile a program behavior for optimization purposes and are available for any application in the user space. These events are also used in the detection of abnormal behaviors in computer systems [99, 115, 114]. We distinguish two different methods of detection: (1) signature based [91, 28] and (2) threshold-based [19]. Signature-based approaches create the signature of the attack based on received information from HPCs, and compare the behavior of the system with the generated signature to identify any eventual malicious activity. For example, Payer et al. [91] proposed an approach to detect cache-based side-channel attacks between two processes. Their approach is based on HPCs and kernel events, e.g., number of page faults, to generate signatures for cache-based side-channel attacks. To detect a malicious process, they compare the generated signatures with the process signature.

It has been shown that resource usage correlates to the probability that an application is of a specific type. For an example, it becomes clear that cache activity is a very strong indicator of this attack type, with applications with a very high level-1 instruction cache (L1-i) and high LLC pressure corresponding to cache based side channel attack with a high probability. Disk traffic also conveys a lot of information, with zero disk usage signaling a side channel attack with very high likelihood. Correlating similarity concepts to resources shows that certain resources are more prone to leaking information about a workload, and their isolation should be prioritized.

On the other hand, threshold-based approaches utilize the HPCs trace to flag anomaly

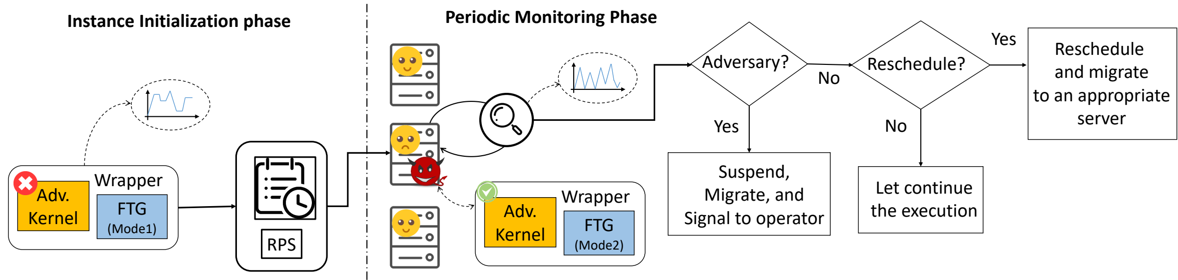

Figure 7.1: Overview of Cloak & Co-locate

resource utilization that goes beyond a pre-specified threshold. For example, Chiappetta et al. [19] proposed to collect certain statistics through HPCs for running processes, and then use machine learning techniques (neural networks and unsupervised learning) to detect a side-channel attack between two processes. Recent works suggest to use machine learning for Malware or side-channel attack detection [37, 117, 113, 118, 98, 101].

Proposed Approach

In any resource sharing-based attack, the instance initialization phase of RPS plays a preventive role in such attacks by avoiding a malicious instance to be co-localized with a targeted victim instance on the same physical machine. This phase of VM placement algorithm is undocumented for security reasons by cloud service providers. However, we show how an adversary can bypass the instance initialization phase and gets co-located by victims with high probability. Moreover, periodic monitoring and rescheduling is leveraged to mitigate co-residency attacks, when the attack is detected. Therefore, we will also show how it is possible to disguise the malicious behavior of adversary's VM and remain on the same host with the victim and avoid the migration.

In this work, we propose to use a *Cloak*, which is technically a fake trace generator (FTG), to wrap it around an adversary application in order to first get *co-located* with a targeted victim and subsequently evade detection and migration. At instance initialization phase, the *Cloak* goal is to perform a trace mimicry task that will mimic the behavior of the targeted victim application to increase the chance of co-location. If the desired server has not been

assigned to the adversary VM (co-residency with the victim can be detected by network probing) then the adversary VM terminates, and a new VM must be reinstantiated as the cost of forcing RPS to migrate the adversary VM to another host that may have the victim is high. After co-residency, the *Cloak* will change its mode to constantly generate a new specialized trace that changes the behavior of the adversary application to not only evade detection but also migration.

To create the *Cloak*, i.e., FTG, we use the concept of adversarial sample in machine learning where we can add specially crafted perturbations to an input signal to fool the machine learning models. In particular, our goal is to change the model's output decision to a specific output, i.e., output that is similar to the targeted victim output. For this purpose, our attack is performed in two phases. First, we need to reverse engineer the resource provisioning system using a machine learning model to have access into to how the RPS make decisions. Second, we utilize the reversed engineering results to craft specialized *Cloak*; a FTG that will add perturbations to the adversary's application trace to force the RPS to co-locate it with a targeted victim. Figure 7.1 shows the overview of our proposed method. The details of this approach will be discussed in the following sections.

7.3 Reverse engineering of RPS

To perform any resource sharing-based attack, such as side-channel attacks, an attacker must co-locate an adversary instance, e.g., an attacker VM with the victim VM. To find hosts that victim's VM are running on, we propose to reverse engineer the resource provisioning system in the IaaS cloud. As mentioned, RPS can be considered as a blackbox (worst-case scenario). Figure 7.2 shows an overview of our reverse engineering scheme. Thus, as a first step, we propose the following methodology to perform the reverse engineering.

The goal of reverse engineering of RPS is to create an ML model that can mimic the functionality of the original RPS. For this purpose, we train an arbitrary ML model, i.e., a proxy model, that can provision a server with the same configuration that the original RPS

provisions for an incoming application to the cloud. We implement PARIS [127], a RPS proposed at UC Berkeley, to act as our original RPS, i.e., victim.

PARIS uses a machine learning technique (Random Forest) for predicting the performance from the application's fingerprint (microarchitectural signature) to find the best VM type configuration. To generate the fingerprint of an application, PARIS extracts 20 resource utilization counters spanning the following broad categories and calls it the fingerprint: CPU utilization, Network utilization, Disk utilization, Memory utilization, and System-level features. On the other hand, CPU count, core count, core frequency, cache size, RAM per core, memory bandwidth, storage space, disk speed, and network bandwidth of the server are the representation of the configuration provisioned by PARIS.

We denote the microarchitectural fingerprint and system level information of an application as $Fing$ vector. In Equation (7.1), a_i denotes each architectural feature.

$$Fing = \{a_1, a_2, ..., a_{20}\} \tag{7.1}$$

configuration parameters of the server's platform referred to configuration inputs is as follow:

$$Conf = \{c_1, c_2, ..., c_9\} \tag{7.2}$$

where $Conf$ is the configuration vector and c_i is the value of the ith configuration parameter (number of sockets, number of cores, core frequency, cache size, memory capacity, memory frequency, number of memory channels, storage capacity, storage speed, network bandwidth).

The RPS is responsible to provision $Conf$ based on $Fing$:

$$Conf = f(Fing) \tag{7.3}$$

Note that $f(Fing)$ is just a data model, which means there is no direct analytical equation to formulate it.

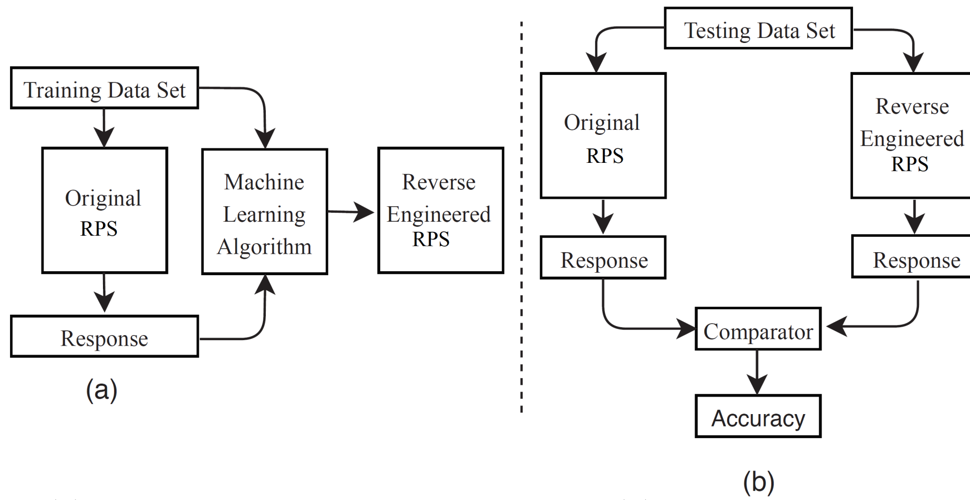

Figure 7.2: (a) Process of reverse engineering RPS; (b) Testing Performance of Reverse-Engineered RPS

Data collection and model training

To use an arbitrary machine learning model to act as a resource provisioning system, we need a training dataset to train a proxy model. The dataset has two parts: the first part is the application's fingerprint and the second part is the corresponding configuration provisioned by RPS. Then we can use the dataset for training our proxy model to map those fingerprints to final configurations.

We perform the data collection in a controlled environment, where all applications are known. We use a medium size 40-machine cluster (presented in Table 5.1), and schedule a total of 120 workloads, including batch analytics in Hadoop and Spark and latency-critical services, such as webservers, Memcached and Cassandra. For each application type, there are several different workloads with respect to algorithms, framework versions, datasets, and input load patterns. The training set is selected to provide sufficient coverage of the space of resource characteristics. Figure 7.3 shows the coverage of resource characteristics for applications in the training set. The selected workloads cover the majority of the resource usage space. This enables us to match any new application that has not been previously seen.

We submit all of these applications to the targeted RPS. In the beginning, the RPS profiles the application and extracts the fingerprint. Then, the RPS uses the Random Forest model to determine an appropriate server configuration. We collect all the fingerprints and their correspondent configurations generated by the RPS to shape our dataset.

We then use an Artificial Neural Network (ANN) to map the $Fing$ to $Conf$. We exploit one ANN per each c_i in the $Conf$. Therefore, we totally train 9 ANNs that each of them has 20 inputs and one output. We started with a simple 5 fully connected layers neural network. We found out this model achieves our desired accuracy. Therefore, we did not use a more complex model, such as a deep neural network. Each ANN has 230 hidden neurons. The number of neurons for the hidden layer is decided through Grid Search [104] to reach the highest possible accuracy.

Performance of reverse engineered RPS

The testing set consists of 108 diverse applications that include web servers, various analytics algorithms and datasets, and several key-value stores and databases. Note that there is less than 30% overlap between training and testing sets in terms of algorithms, datasets, and input loads. The Original RPS is fed with all applications and the responses are recorded. These responses are utilized in order to compare the functionality of ANNs versus the original RPS. We observed that ANNs perform well with an overall accuracy of 92.7% to mimic the original RPS, the precision of 0.90, recall 0.93 and F1-score of 0.91. Now, we have a proxy model in hand that can be utilized to generate adversarial.

7.4 Cloak Generator

Our proposed *Cloak*, i.e., FTG, works in two modes. Mode (1) is for the instant initialization phase (before co-location) and mode (2) is for the periodic monitoring phase. When an adversary application is submitted to the RPS, the adversary kernel is deactivated. However,

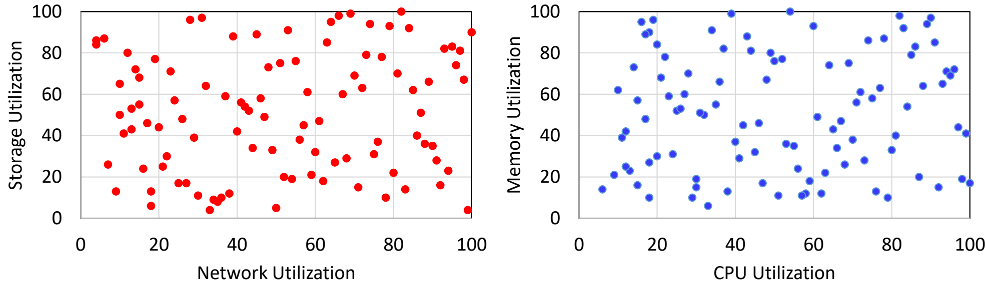

Figure 7.3: Coverage of application's characteristic

FTG must start working and change the behavior during instant initialization phase to generate a trace similar to the victim application in order to fool the RPS and get co-located with the victim on the same host.

After co-location, when co-residency has been detected by using any techniques mentioned in Section 7.2, the adversary kernel can starts its attack. While starting the attack, the *Cloak*, i.e., FTG, switches to mode (2), where it is required to carefully generate a fake trace that disguises the behavior of the adversary kernel. Moreover, this trace must fool the RPS to provision the same configuration as it provisioned during the instant initialization phase. For this purpose, FTG must constantly monitor the system's state. In order to monitor the system state and extract the Hardware performance counters (HPC) information, FTG uses $Perf$ tool available under Linux. Perf provides rich generalized abstractions over hardware specific capabilities. It exploits $perf-event-open$ function call in the background which can measure multiple events simultaneously. It is non-trivial to determine the level of perturbations that need to be injected into the application's microarchitectural patterns in order to get the desired host configuration. Crafting a trace that can fool the RPS is only possible by performing a targeted adversarial attack on RPS. We discuss this targeted adversarial attack in the next subsection of the paper.

When the trace generated by FTG has been calculated (either in mode (1) or mode (2)), FTG runs a few micro benchmarks of tunable intensity from iBench [24] that each put pressure on a specific shared resource. iBench consists of a set of carefully-crafted benchmarks that generate contention on core, the cache and memory hierarchy, and the

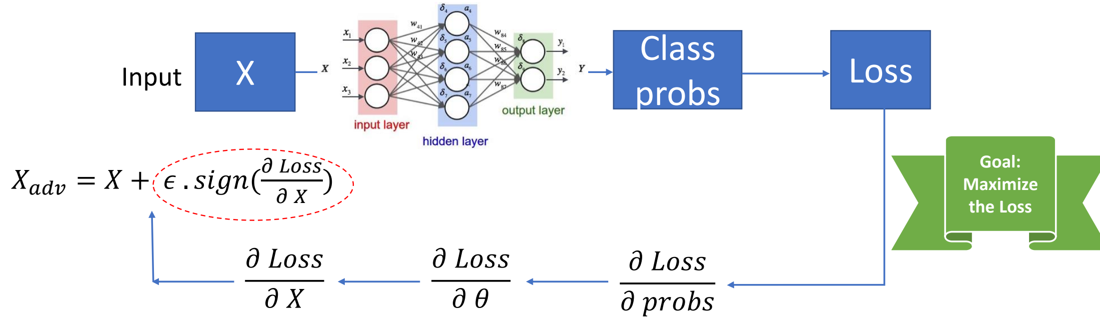

Figure 7.4: Fast Gradient Sign Method

storage and networking subsystems.

Targeted adversarial attack to RPS

To perturb the microarchitectural patterns, we employ a low-complex gradient loss based approach, similar to Fast-Gradient Sign Method (FGSM), which is widely employed in image processing. The advantage of this approach is its low complexity and low computational overheads. In order to craft the adversarial perturbations, we consider the proxy model generated from the reverse engineering of the targeted RPS. In our experiments, the proxy model is a neural network with θ as the hyperparameters, x being the input to the model (current fingerprint of adversary kernel), and y is the output (current configuration of server) for a given input x, and $L(\theta, x, y)$ be the cost function used to train the neural network. Then the perturbation required to change the output to the target configuration is determined based on the cost function gradient of the neural network (in this case). Eventually, this perturbation is the trace that must be generated by FTG. The adversarial perturbation is calculated based on the gradient loss, as shown in Figure 7.4, similar to the FGSM [30] and is given by:

$$x^{adv} = x + \epsilon \; sign(\nabla_x \; L(\theta, x, y)$$

where ϵ is a scaling constant ranging between 0.0 to 1.0 and is set to be very small such that the variation in $x(\delta x)$ is undetectable. In case of FGSM, the input x is perturbed along each dimension in the direction of gradient by a perturbation magnitude of ϵ. Considering

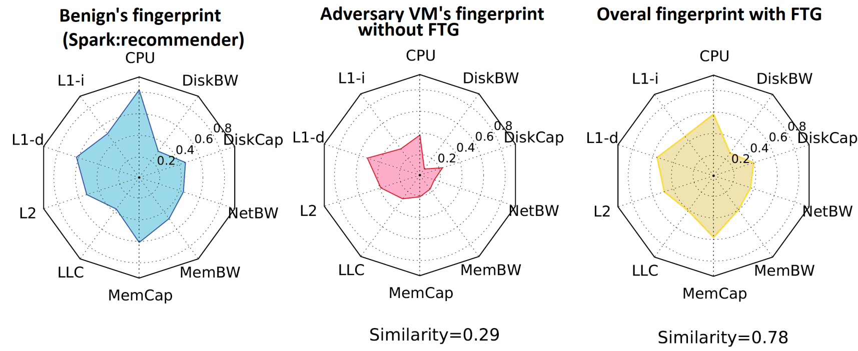

Figure 7.5: Activation of FTG and increase in the similarity of fake trace and victim's fingerprint during instant initialization phase

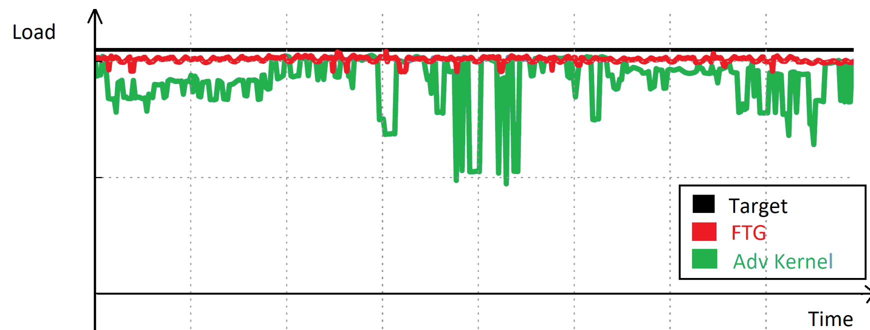

Figure 7.6: Overall trace using FTG, adversary kernel trace, and the target

a small ϵ leads to well-disguised adversarial samples that successfully fool the ML model. In contrast to the images where the number of features is large, the number of features, i.e., microarchitectural metrics are limited; thus, the perturbations need to be crafted carefully and, more importantly, can be generated during runtime by the applications. For instance, a negative value cannot be generated by an application. Hence, we provided a lower bound on the adversary values.

Figure 7.5 shows a side channel attack and its fingerprint's similarity with a victim application (Spark: recommender system) during instant initialization phase. After the activation of FTG, the similarity of fingerprint increases to 78% and therefore the adversary VM can be identified as friendly VM. Figure 7.6 illustrates how FTG can generate the desired trace proposed by adversarial sample generator during periodic monitoring phase (when co-residency has been detected). The blue line shows the trace of adversarial kernel.

The target trace is the black line, which shows the overall trace must be the same to be able to evade detection and migration. The red line shows the overall trace (with the impact of FTG trace), which is the trace after adding perturbation. We can observe that the fake trace generator is successful in disguising the adversarial kernel.

7.5 Evaluation

To evaluate our proposed approach, we implemented 8 distributed attacks as follow: SC1: Prime + Probe, SC2: Flush + Reload, SC3: Flush + Flush, SC4: Evict + Time, DoS1: increasing latency by saturating the network, DoS2: decreasing throughput by saturating storage, RFA1: freeing memory resource, and RFA2: freeing CPU resource. We perform these attacks on 20 unseen victim applications from different domains (SPEC, Hadoop, Spark, Memcache, and Cassandra). Based on our evaluation, the success rate of being co-located with victims, evading the detection and migration, and getting the desired outcome from attack depends on many factors such as victim's type, the period of monitoring phase, and amount of perturbation.

Experimental results

Table 7.1 shows the impact of victims' type on the success rate of each type of attack. The interesting observation is that there is a meaningful relationship between the application's type and the nature of the attack by itself. For instance, we observe that side-channel attacks are more successful when the cache hit rate of the victim is low. Similarly, we observed that RFA is more successful when the resource utilization of the victim is high. One reason is that in such case, FTG can generate a better fake trace to convince the RPS to stay at the current host. In a case that the difference between the behavior of the adversary kernel and the victim is high, the FTG has to generate more perturbation and this may lead to a migration decision by RPS.

Table 7.1: Success Rate (SR) of distributed attacks based on the application's type. (*: SR $\leq 25\%, ** : 25\% < SR \leq 75\%, *** : SR \geq 75\%$)

	SPEC	Hadoop	Spark	Memcached	Cassandra
SC1	***	*	**	*	**
SC2	***	*	*	*	**
SC3	***	*	**	**	*
SC4	***	**	**	*	**
DoS1	*	*	***	***	**
DoS2	*	***	*	*	***
RFA1	*	**	***	***	**
RFA2	***	**	***	***	*

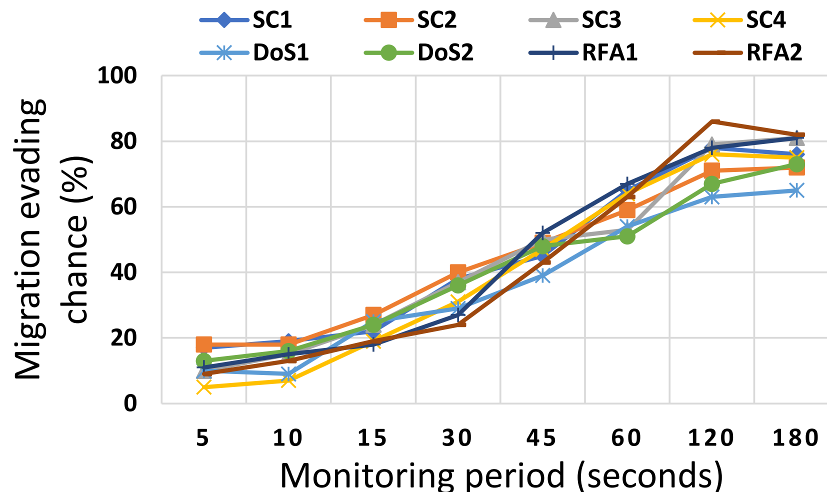

Figure 7.7: Impact of monitoring period on the chance of evasion from migration

In order to provide an insight into the impact of the monitoring period on the evasion chance of attack and the effectiveness of FTG, we performed each attack under different monitoring periods. Figure 7.7 shows the results of this experiment. One can observe that by increasing the period, the chance of evasion increases. The reason is that the fingerprint of application changes more frequently when the period is small and FTG is not able to generate the desired trace. However, when the period increases, the perturbation that FTG generates becomes more effective.

It is important to know how much perturbation is required to evade the detection but still have a good chance for remaining on the same host and evade migration. Figure 7.8

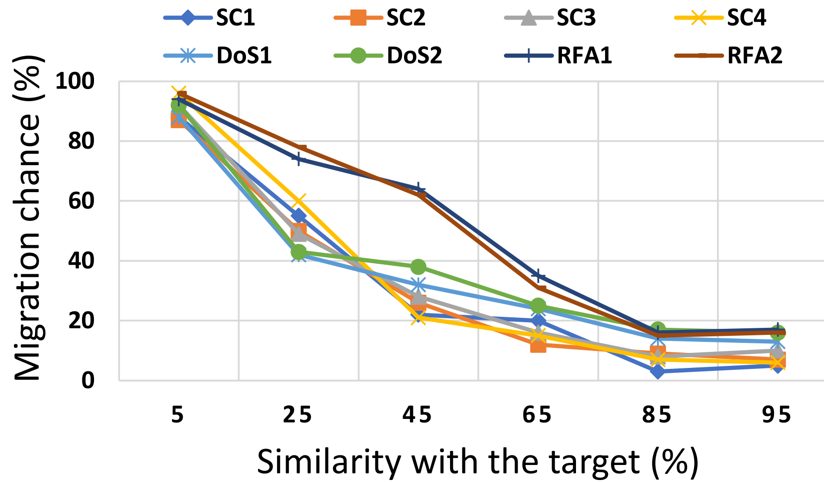

Figure 7.8: Impact of similarity with the target trace on migration chance

shows the average migration chance of each attack and the similarity of our adversary VM's behavior with the victim's behavior. It was expected that by increasing the similarity between the adversary and victim, the migration chance reduces. However, we observed that generating fake trace to exactly follow the target trace proposed by adversarial sample is hard. Therefore, reaching to 0% chance of migration is out of reach. Moreover, Figure 7.9 demonstrates the impact of the perturbation amount on the migration chance. The results show that the amount of perturbation should be carefully determined as a high or low amount of perturbation causes migration.

We evaluated the impact of perturbation on the effectiveness of the attack by itself and obtaining the desired outcome. As Figure 7.10 shows, side-channel attacks are very sensitive to a perturbation such that by increasing the perturbation, the attacks become useless. This is due to the fact that side channel attacks are customized and fine-grained and any noise on the system, especially on the hardware performance counters, can prevent the attack from being successful.

Last but not least, we evaluated the number of VM's per host and its impact on the success rate of Cloak & Co-locate attack. We run randomly multiple friendly applications from our application pool on each host and perform an attack on one of them. The success

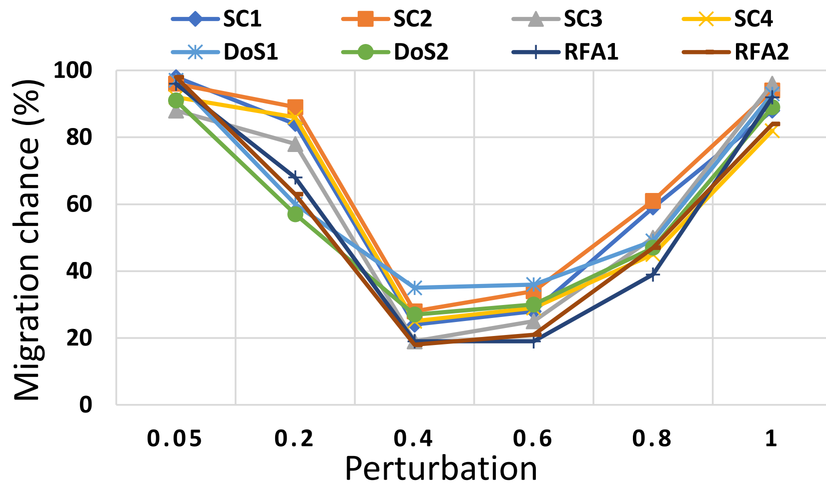

Figure 7.9: Impact of perturbation on migration chance

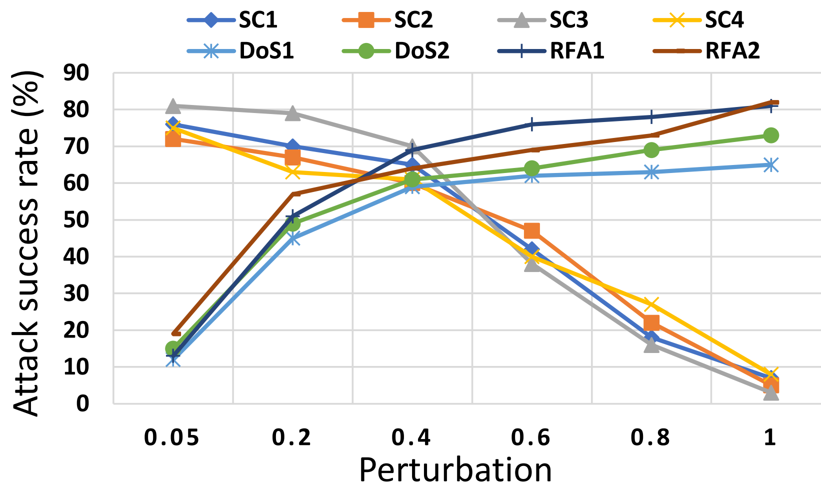

Figure 7.10: Impact of perturbation on attack success rate

rate of attacks mostly depends on the server's type as the amount of resources available on the host impacts the applications' fingerprints, and configuration assignment. Figure 7.11 demonstrates the average success rate of attacks corresponding to the number of friendly VMs on the same host. The trend shows that increasing the number of applications on the host significantly reduces the attack's success rate from 60% to 16% on average.

The breakdown of attacks' success or failure rate on a general purpose server is presented in Figure 7.12. The red bar shows the percentage of failing to co-locate with victim at the instantiate phase. The gray bar shows that the adversary co-located with the victim

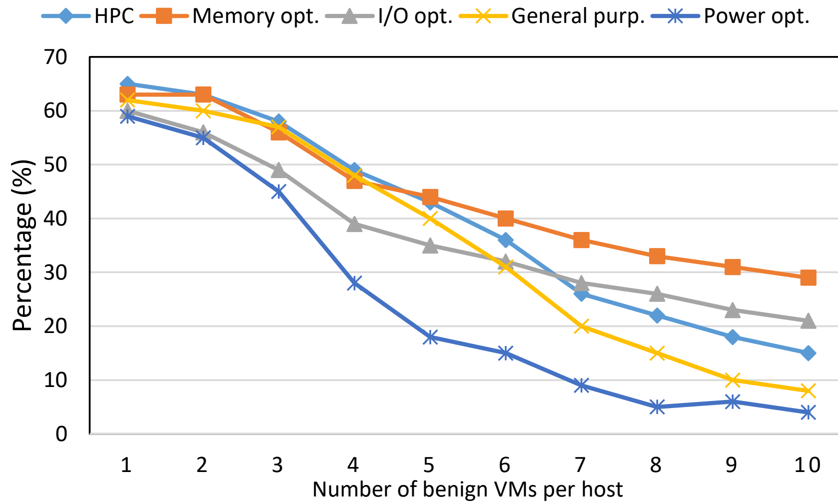

Figure 7.11: Average of attacks' success rate when multiple VMs are running in the host

at the instantiate phase but RPS migrated the adversary VM to another host during the monitoring phase. The blue bar shows the percentage that the adversary co-located with the victim and remained co-locate on the same host and was not detected or migrated. However, the adversary could not perform the attack successfully. The green bar shows that the attack was completely successful. This figure shows that by increasing the number of VMs per host, the chance of migration or not being co-locate by the victim increases up to 4X. It shows even if the adversary co-locates with the victim, the chance of success would be low as there are other applications running on the host that impacts the attack's setting.

Improving security using resource isolation

In order to study the effects of resource isolation, we enforce several resource partitioning and isolation techniques discussed previously in section 6.2. We employ core isolation (thread pinning to physical cores), to constrain interference context switching. We employ the Cache Allocation Technology (CAT) available in Intel chips [49] to isolate last level cache (LLC). The size of cache partitions can be changed at runtime by reprogramming MSR registers. We also use the outbound network bandwidth partitioning capabilities of Linux's traffic control.

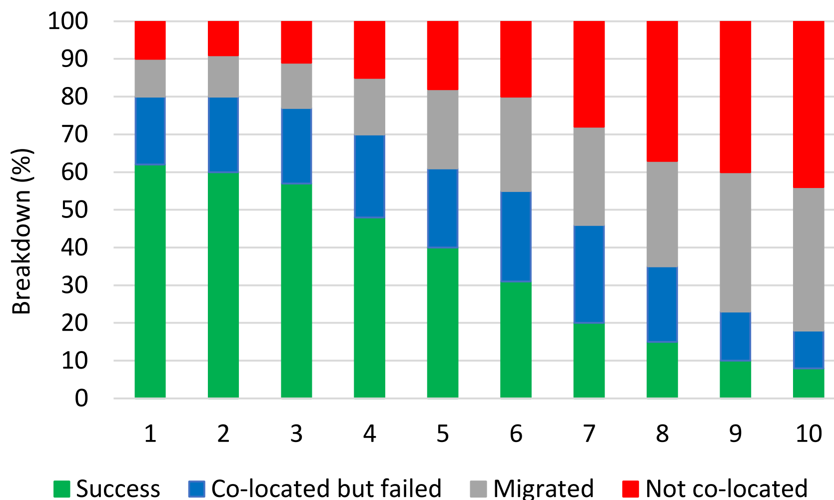

Figure 7.12: Breakdown of attacks' success or failure on a general purpose server

We employ the `qdisc` [77] to enforce bandwidth limits. To perform DRAM bandwidth partitioning, RPS monitors the DRAM bandwidth usage of each application using Intel PCM to co-locate jobs on the same machine where it can accommodate their aggregate peak memory bandwidth usage.

Figure 7.13 shows the impact of isolation techniques on the effectiveness of attacks. As expected, when no isolation is used, we have a significantly higher success rate. As a result, introducing thread pinning benefits, since it reduces core contention. The dominant resource usage of each application determines which isolation technique benefits it the most. Thread pinning mostly benefits workloads bound by on-chip resources, such as L1/L2 caches and cores. Adding network bandwidth partitioning lowers success rate for DoS attacks. It primarily benefits network-bound workloads, for which network interference conveys the most information for detection of co-residency. Cache partitioning has the most dramatic reduction in success rate of SCA, as they are LLC-bound applications. Enforcing core isolation is also sufficient to degrade the success rate of RFAs. Finally, memory bandwidth isolation further reduces success rate by 10% on average, benefiting jobs dominated by DRAM traffic. It is more effective on DoS and RFA and has less impact on SCAs.

The number of co-residents also affects the extent to which isolation helps. The more

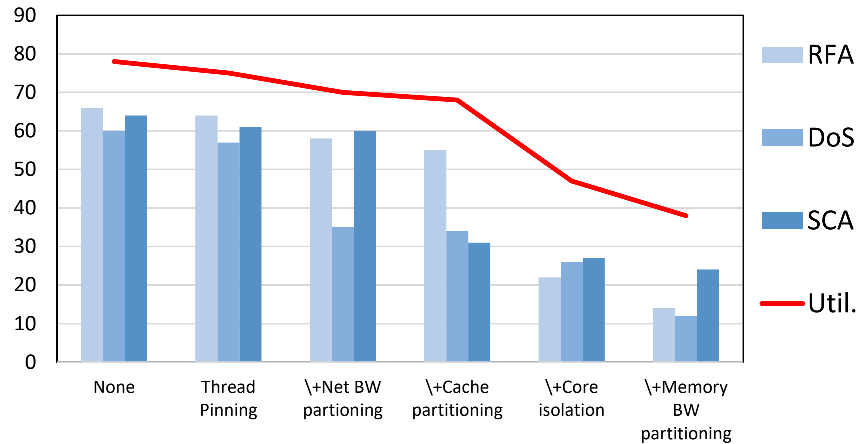

Figure 7.13: Attacks' success rate and CPU utilization (Util) with isolation techniques

co-scheduled applications exist per machine, the more isolation techniques degrade success rate, as they make distinguishing between co-residents harder. Improving security using isolation, however, comes at a performance penalty of 32% on average in execution time, as threads of the same job are forced to contend with one another. Alternatively, users can overprovision their resource reservations to avoid performance degradation, which results in a 47% drop in utilization. This means that the cloud provider cannot leverage CPU idleness to share machines, decreasing the cost benefits of cloud computing.

Our analysis highlights a design problem with current datacenter platforms. Traditional multicores are prone to contention, which will only worsen as more cores are integrated into each server, and multi-tenancy becomes more pronounced. Existing isolation techniques are insufficient to mitigate security vulnerabilities, and techniques that provide reasonable security guarantees sacrifice performance or cost efficiency, through low utilization. This highlights the need for new techniques to be invented that guarantee security at high utilization for shared resources.

Transferability analysis

Different RPSs use different underling mechanisms to preform resource provisioning. For instance, Quasar [26] leverages collaborative filtering to quickly determine which applications can be co-scheduled on the same machine without destructive interference. CherryPick [2] and Ernest [112] also uses machine learning for selecting close to optimal configuration on the cloud. Therefore, we tried to answer the question of whether an adversarial *Cloak* that is targeting a specific RPS can also force other RPS, which uses different underling mechanisms, to co-locate it with the victim. Specifically, we are testing the transferability of our attack methodology across RPS, which will showcase the robustness of the proposed adversarial sample crafting.

For this purpose, we setup an experiment where we generate *Cloaks* that target a specific RPS and see if they can also fool the other RPS. Table 7.2 shows the results of our transferability analysis for all combinations. It should be noted that the analysis is performed under these circumstances: perturbation is set to 0.4, the monitoring period is one minute, the number of VM allowed on the same host is two, and none of the isolation techniques is used. Each value of table 7.2 is attacks' average success rate (in percentage). Each row represents the original RPS and each column represents the substituted RPS. Where the column and the row are equal, it means we did not replace the RPS. For example, we used PARIS as our original RPS and then we reverse-engineered it. For transferability analysis, we replaced the original RPS (PARIS) with another RPS (CherryPick in this example). Then we used the Cloak & Co-locate approach to attack the system. Afterward, we evaluated the success rate of our attacks. In the above example, the average success rate of attacks was 56.5% as shown in Table 7.2.

The results from Table 7.2 show that the reverse-engineered model from PARIS is more generalizable than others as the transferability of attacks is 55.1% (shown in blue color). Because of this generalization, we selected PARIS as our original RPS to study the Cloak & Co-locate attack throughout the paper. The results also show that the traces generated

Table 7.2: Attacks' average success rate (all values are percentage). Each row represent original RPS and each column is for substituted RPS

	PARIS	**CherryPick**	**Ernest**	**Quasar**	**Average**
PARIS	62.3	56.5	53.2	48.7	55.1
CherryPick	54.4	65.4	54	44.1	54.4
Ernest	46.8	55.7	67.4	33.5	50.8
Quasar	53.6	47.3	52.9	59.3	53.3
Average	54.3	56.2	56.9	46.4	63.6

by FTG for PARIS can be applied to CherryPick, Ernest, and Quasar with a success rate of 56%, 53%, and 48%, respectively. This indicates that the ML model used to craft the adversarial samples is transferable to other systems as long as we can mimic the RPS's functionality.

On the other hand, results show that Ernest has the lowest transferability as it is the simplest RPS in our evaluation, and the average success rate of attacks transferred from Ernest to other RPSs is 50.8%. Ernest is based on a statistical technique and therefore cannot capture the complexity of the system. For the same reason, it is the most vulnerable RPS as the transferability of attacks from other RPSs to it is the highest (average success rate is around 56.9%, shown in red color). We observed that Quasar is the most resilient RPS against Cloak & Co-locate attack. The average success is only 46.4% as shown in the green color. The Quasar's collaborative filtering and its complexity are the reasons behind this immunity. Without considering the transferability (no substitution of original RPS), the average success rate of attacks is around 63% (the purple color).

7.6 Conclusions

In this work, we proposed Cloak & Co-locate – a novel approach to improve the effectiveness of distributed attacks on cloud infrastructure. For this purpose, we demonstrated that by reverse-engineering the resource provisioning system and employing the adversarial machine learning attack, adversary VM can co-locate itself with the victim and evade detection, as

well as migration caused by the scheduler. For this purpose, we proposed to use a fake trace generator (Cloak) and wrap it around the adversary kernel. The fake trace generator is spawned as a separate thread, generating a pattern close to the victim VM's pattern, fooling the scheduler to co-locate it with the victim VM. After co-location, FTG continuously crafts new behavior to disguise itself and fool RPS for remaining co-located on the same host as the victim. Our results show that applying strict isolation can reduce the impact of such attacks while increases the cost of cloud (lower utilization). This research work, while deployed on a medium-size cluster, will motivate real-world public cloud providers to introduce stricter isolation solutions in their platforms and systems architects to develop robust RPSs that provide security and performance predictability at high utilization.

Chapter 8

Future Directions and Conclusion

In this dissertation, we argued for machine learning-based resource provisioning systems for data-intensive applications in scale-out platforms.

Through our empirical performance evaluation (Chapter 3), we provided an insight into the role the memory subsystem plays in the overall performance of the servers when running data analytics frameworks. Characterizing the memory behavior of frameworks is important as it helps to guide scheduling decisions in cloud-scale architectures as well as helping to make decisions in designing better clusters for data-intensive applications.

We introduced MeNa in chapter 4 to help memory provisioning for data-intensive applications and increase the performance/cost.

With E-Net (Chapter 5), we designed, implemented, and evaluated a predictive scheduler for improved and predictable job completion times while reducing total resources used and power consumption. E-Net addresses the challenge of resource provisioning for IMC workloads in heterogeneous scale-out platforms consist of diverse types of servers.

In chapter 6, we utilized what we have developed in MeNa and E-Net to adaptively model the performance of data-intensive applications and use it for developing a highly accurate RPS, called ProMLB.

The security concerns rising from ML-Based RPSs have been studied in chapter 7. We

showed that by reverse-engineering the resource provisioning system and employing the adversarial machine learning attack, adversary VM can co-locate itself with the victim and evade detection, as well as migration caused by the scheduler. One of the mitigation for such vulnerability is to use an isolation mechanism and we evaluated the effectiveness of this approach in reducing the risk of ML-based RPSs.

There is a lot left to be done to achieve a secure high-utilization automatic resource provisioning system. moreover, a lot of new avenues of research will open up as we move forward in using machine learning techniques. Before the conclusion, we present upcoming and potential future directions.

8.1 Avenues for Future Directions in ML Research

With the evolution of computing, infrastructures, and frameworks, we see more and more decentralization. This trend can include the provisioning of resources that we call it federated resource provisioning system era. Federated RPS can have many attractive features, such as no need to manage individual servers, continuous scaling, and sub-second metering. However, this approach can simultaneously reduce users' control over the underlying physical resources. Exploring this research direction needs to deal with many challenges including understanding performance and power as a function of allocated resources, and deciding on the granularity of federation to allow predictability of performance or any other optimization target.

One important direction that we have not studied in this dissertation is the use of GPUs and FPGAs in modern datacenters for accelerating the performance of data-intensive applications. Another interesting direction to continue this research is to model the performance of such hardware accelerators and integrate them in RPSs. The current setup of our work can easily get expanded and be used in future works.

Machine Learning techniques are being exploited in various applications, in addition to the field of resource provisioning discussed in this dissertation, in which the data continuously is collected. There are two important questions in each ML application; when and how to

retrain ML models as the environment changes and how to guarantee that the collected data is appropriate. Further work is required while advances in online learning, and reinforcement learning address different aspects of this issue. Because the integration of these techniques with existing RPSs and retraining algorithms can be challenging.

In addition to accurate predictions, when applying ML techniques for deriving decisions in the domain of RPSs, it is important for experts and designers to obtain insights from the decision-making process of the ML models. The need for explainability is the reason behind selecting simple models such as decision trees in the current model-driven systems. Powerful, popular, and complex models such as deep learning-based models, are not easy to understand. Another new avenue to further ML-based RPSs is to build models that achieve both, high accuracy and interpretability. Exploring interpretable ML models for RPS is one of our future works.

8.2 Concluding Remarks

In this dissertation, we demonstrated the effectiveness of machine learning models for developing high-accurate and high-utilization resource provisioning systems for data-intensive applications in scale-out platforms, by dealing with the challenges they bring, such as security vulnerability, model uncertainty, generalization from benchmark suits to real applications, high-cost training, and interpretability of existing ML models. We discussed issues that are left behind to be addressed including the challenge of performance modeling of hardware accelerators, and developing more interpretable ML models. It is essential to comprehend when we should use machine learning models in the context of systems. Based on our understanding, machine learning should be deployed when the size or the complexity of the problem dominates the decision-making process. Heuristic approaches were used in several computing systems and they were suitable for the common cases. In the era of heterogeneous datacenters and with the growing complexity of frameworks, we are facing situations that require custom decisions. Machine learning, unlike heuristics, is an appropriate solution for

new challenges. Moreover, while we have focused on improving resource efficiency at the cluster management level, designing novel hardware and software schemes that ensure strict isolation between applications can improve resource efficiency too. We leave these endeavors to future work.

Bibliography

[1] Sanjay P Ahuja, Thomas F Furman, Kerwin E Roslie, and Jared T Wheeler. "Empirical Performance Assessment of Public Clouds Using System Level Benchmarks". In: *International Journal of Cloud Applications and Computing (IJCAC)* 3.4 (2013), pp. 81–91.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. "Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics". In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 469–482.

[3] Albino Altomare, Eugenio Cesario, and Andrea Vinci. "Data Analytics for Energy-Efficient Clouds: Design, Implementation and Evaluation". In: *International Journal of Parallel, Emergent and Distributed Systems* 34.6 (2019), pp. 690–705.

[4] Icaro Alzuru, Kevin Long, Bhaskar Gowda, David Zimmerman, and Tao Li. "Hadoop Characterization". In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 2. IEEE. 2015, pp. 96–103.

[5] *Available at: http://jenetics.io/*.

[6] *Available at: https://www.mathworks.com/products-/matlab.html*.

[7] Sean Kenneth Barker and Prashant Shenoy. "Empirical Evaluation of Latency-Sensitive Application Performance in the Cloud". In: *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*. 2010, pp. 35–46.

[8] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. "Memory System Characterization of Commercial Workloads". In: *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*. IEEE. 1998, pp. 3–14.

[9] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. "Efficient Virtual Memory for Big Memory Servers". In: *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 237–248.

[10] Mohammad-Mahdi Bazm, Marc Lacoste, Mario Südholt, and Jean-Marc Menaud. "Isolation in Cloud Computing Infrastructures: New Security Challenges". In: *Annals of Telecommunications* 74.3-4 (2019), pp. 197–209.

[11] Scott Beamer, Krste Asanovic, and David Patterson. "Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server". In: *2015 IEEE International Symposium on Workload Characterization*. IEEE. 2015, pp. 56–65.

[12] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. "RFHOC: A Random-Forest Approach to Auto-Tuning Hadoop's Configuration". In: *IEEE Transactions on Parallel and Distributed Systems* 27.5 (2015), pp. 1470–1483.

[13] Elisa Bertino. "Big Data–Opportunities and Challenges Panel Position Paper". In: *2013 IEEE 37th Annual Computer Software and Applications Conference*. IEEE. 2013, pp. 479–480.

[14] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM. 2008, pp. 72–81.

[15] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. "Evasion Attacks Against Machine Learning

at Test Time". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2013, pp. 387–402.

[16] Peter Bodik, Rean Griffith, Charles Sutton, Armando Fox, Michael I Jordan, and David A Patterson. "Automatic Exploration of Datacenter Performance Regimes". In: *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*. 2009, pp. 1–6.

[17] Claudio Canella, Khaled N Khasawneh, and Daniel Gruss. "The Evolution of Transient-Execution Attacks". In: *Proceedings of the 2020 on Great Lakes Symposium on VLSI*. 2020, pp. 163–168.

[18] CL Philip Chen and Chun-Yang Zhang. "Data-Intensive Applications, Challenges, Techniques and Technologies: A Survey on Big Data". In: *Information Sciences* 275 (2014), pp. 314–347.

[19] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. "Real Time Detection of Cache-Based Side-Channel Attacks Using Hardware Performance Counters". In: *Applied Soft Computing* 49 (2016), pp. 1162–1174.

[20] Sanket Chintapalli et al. "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming". In: *IEEE Parallel and Distributed Processing Symposium Workshops*. 2016.

[21] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. "Quantifying the Performance Impact of Memory Latency and Bandwidth for Big Data Workloads". In: *2015 IEEE International Symposium on Workload Characterization*. IEEE. 2015, pp. 213–224.

[22] Christina Delimitrou and Christos Kozyrakis. "Bolt: I Know What You Did Last Summer... in the Cloud". In: *ACM SIGARCH Computer Architecture News*. Vol. 45. 1. ACM. 2017, pp. 599–613.

[23] Christina Delimitrou and Christos Kozyrakis. “Hcloud: Resource-Efficient Provisioning in Shared Cloud Systems”. In: (2016), pp. 473–488.

[24] Christina Delimitrou and Christos Kozyrakis. “iBench: Quantifying Interference for Datacenter Applications”. In: *IISWC*. IEEE. 2013.

[25] Christina Delimitrou and Christos Kozyrakis. “Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters”. In: *ACM SIGPLAN Notices* 48.4 (2013), pp. 77–88.

[26] Christina Delimitrou and Christos Kozyrakis. “Quasar: Resource-Efficient and QoS-Aware Cluster Management”. In: *ACM SIGARCH Computer Architecture News*. Vol. 42. 1. ACM. 2014, pp. 127–144.

[27] Martin Dimitrov, Karthik Kumar, Patrick Lu, Vish Viswanathan, and Thomas Willhalm. “Memory System Characterization of Big Data Workloads”. In: *2013 IEEE International Conference on Big Data*. IEEE. 2013, pp. 15–22.

[28] Sai Manoj Pudukotai Dinakarrao, Hossein Sayadi, Hosein Mohammadi Makrani, Cameron Nowzari, Setareh Rafatirad, and Houman Homayoun. “Lightweight Node-Level Malware Detection and Network-Level Malware Confinement in IoT Networks”. In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 776–781.

[29] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. “Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware”. In: *ACM SIGPLAN Notices*. Vol. 47. 4. ACM. 2012, pp. 37–48.

[30] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. “Explaining and Harnessing Adversarial Examples”. In: *arXiv preprint arXiv:1412.6572* (2014).

[31] Lei Gu and Huan Li. "Memory or Time: Performance Evaluation for Iterative Operation on Hadoop and Spark". In: *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*. IEEE. 2013, pp. 721–727.

[32] Brian Guenter, Navendu Jain, and Charles Williams. "Managing Cost, Performance, and Reliability Tradeoffs for Energy-Aware Server Provisioning". In: *2011 Proceedings IEEE INFOCOM*. IEEE. 2011, pp. 1332–1340.

[33] Marisabel Guevara, Benjamin Lubin, and Benjamin C Lee. "Navigating Heterogeneous Processors with Market Mechanisms". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2013, pp. 95–106.

[34] Sanchika Gupta and Padam Kumar. "Vm Profile Based Optimized Network Attack Pattern Detection Scheme for DDoS Attacks in Cloud". In: *International Symposium on Security in Computing and Communication*. Springer. 2013, pp. 255–261.

[35] Kevin Gurney. *An Introduction to Neural Networks*. CRC press, 2014.

[36] Damien Hardy, Isidoros Sideris, Ali Saidi, and Yiannakis Sazeides. "EETCO: A Tool to Estimate and Explore the Implications of Datacenter Design Choices on the TCO and the Environmental Impact". In: *Workshop on Energy-efficient Computing for a Sustainable World*. 2011.

[37] Zhangying He, Tahereh Miari, Hosein Mohammadi Makrani, Mehrdad Aliasgari, Houman Homayoun, and Hossein Sayadi. "When Machine Learning Meets Hardware Cybersecurity: Delving into Accurate Zero-Day Malware Detection". In: *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. IEEE. 2021, pp. 85–90.

[38] John L Henning. "SPEC CPU2006 Benchmark descriptions". In: *ACM SIGARCH Computer Architecture News* 34.4 (2006), pp. 1–17.

[39] Herodotos Herodotou. “Hadoop Performance Models”. In: *arXiv preprint arXiv:1106.0940* (2011).

[40] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. “Starfish: A Self-Tuning System for Big Data Analytics”. In: *Cidr*. Vol. 11. 2011. 2011, pp. 261–272.

[41] *https://flink.apache.org*.

[42] *https://hadoop.apache.org/*.

[43] *https://software.intel.com/en-us/articles/intel-performance-counter-monitor*.

[44] *https://spark.apache.org/*.

[45] *https://tez.apache.org/*.

[46] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. “The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis”. In: *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*. IEEE. 2010, pp. 41–51.

[47] Kathlene Hurt and Eugene John. “Analysis of Memory Sensitive SPEC CPU2006 Integer Benchmarks for Big Data Benchmarking”. In: *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*. 2015, pp. 11–16.

[48] Mehmet Sinan Inci, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. “Co-Location Detection on the Cloud”. In: *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer. 2016, pp. 19–34.

[49] *Intel R 64 and IA-32 Architecture Software Developer’s Manual, vol3B: System Programming Guide, Part 2*. 2014.

[50] Joseph Issa. “Performance Characterization and Analysis for Hadoop K-means Iteration”. In: *Journal of Cloud Computing* 5.1 (2016), p. 3.

[51] Keith R Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J Wasserman, and Nicholas J Wright. "Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud". In: *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. IEEE. 2010, pp. 159–168.

[52] Shahram Jamali, Sepideh Malektaji, and Morteza Analoui. "An Imperialist Competitive Algorithm for Virtual Machine Placement in cloud Computing". In: *Journal of Experimental & Theoretical Artificial Intelligence* 29.3 (2017), pp. 575–596.

[53] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. "Characterizing Data Analysis Workloads in Data Centers". In: *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2013, pp. 66–76.

[54] Zhen Jia, Jianfeng Zhan, Lei Wang, Rui Han, Sally A McKee, Qiang Yang, Chunjie Luo, and Jingwei Li. "Characterizing and Subsetting Big Data Workloads". In: *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE. 2014, pp. 191–201.

[55] Tao Jiang, Qianlong Zhang, Rui Hou, Lin Chai, Sally A Mckee, Zhen Jia, and Ninghui Sun. "Understanding the Behavior of In-Memory Computing Workloads". In: *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE. 2014, pp. 22–30.

[56] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. "Profiling a Warehouse-Scale Computer". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 2015, pp. 158–169.

[57] George Kousiouris, Andreas Menychtas, Dimosthenis Kyriazis, Spyridon Gogouvitis, and Theodora Varvarigou. "Dynamic, Behavioral-Based Estimation of Resource Pro-

visioning Based on High-Level Application Terms in Cloud Platforms". In: *Future Generation Computer Systems* 32 (2014), pp. 27–40.

[58] Neeraj Kulkarni, Feng Qi, and Christina Delimitrou. "Leveraging Approximation to Improve Datacenter Resource Efficiency". In: *IEEE Computer Architecture Letters* 17.2 (2018), pp. 171–174.

[59] Palden Lama and Xiaobo Zhou. "Aroma: Automated Resource Allocation and Configuration of MapReduce Environment in the Cloud". In: *Proceedings of the 9th International Conference on Autonomic Computing*. 2012, pp. 63–72.

[60] Pavel Laskov et al. "Practical Evasion of a Learning-Based Classifier: A Case Study". In: *2014 IEEE symposium on security and privacy*. IEEE. 2014, pp. 197–211.

[61] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. "BoPF: Mitigating the Burstiness-Fairness Tradeoff in Multi-Resource Clusters". In: *ACM SIGMETRICS Performance Evaluation Review* 46.2 (2019), pp. 77–78.

[62] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge university press, 2020.

[63] Fan Liang, Chen Feng, Xiaoyi Lu, and Zhiwei Xu. "Performance Characterization of Hadoop and Data MPI based on Amdahl's Second Law". In: *2014 9th IEEE International Conference on Networking, Architecture, and Storage (NAS)*. IEEE. 2014, pp. 207–215.

[64] Luo Lie. "Heuristic Artificial Intelligent Algorithm for Genetic Algorithm". In: *Key Engineering Materials*. Vol. 439. Trans Tech Publ. 2010, pp. 516–521.

[65] Fei Liu, Lanfang Ren, and Hongtao Bai. "Mitigating Cross-VM Side Channel Attack on Multiple Tenants Cloud Platform". In: *Journal of Computers* 9.4 (2014), pp. 1005–1013.

[66] Amiya K Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. "Mitigating Interference in Cloud Services by Middleware Reconfiguration". In: *Proceedings of the 15th International Middleware Conference*. 2014, pp. 277–288.

[67] Hosein Mohamamdi Makrani, Hossein Sayadi, Najmeh Nazari, Sai Mnoj Pudukotai Dinakarrao, Avesta Sasan, Tinoosh Mohsenin, Setareh Rafatirad, and Houman Homayoun. "Adaptive Performance Modeling of Data-intensive Workloads for Resource Provisioning in Virtualized Environment". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 5.4 (2021), pp. 1–24.

[68] Hosein Mohammadi Makrani and Houman Homayoun. "Memory Requirements of Hadoop, Spark, and MPI Based Big Data Applications on Commodity Server Class architectures". In: *2017 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2017, pp. 112–113.

[69] Hosein Mohammadi Makrani and Houman Homayoun. "MeNa: A Memory Navigator for Modern Hardware in a Scale-Out Environment". In: *2017 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2017, pp. 2–11.

[70] Hosein Mohammadi Makrani, Amir Mahdi Hosseini Monazzah, Hamed Farbeh, and Seyed Ghassem Miremadi. "Evaluation of Software-Based Fault-Tolerant Techniques on Embedded OS's Components". In: *International Conference on Dependability (DEPEND)*. 2014, pp. 51–57.

[71] Hosein Mohammadi Makrani, Setareh Rafatirad, Amir Houmansadr, and Houman Homayoun. "Main-Memory Requirements of Big Data Applications on Commodity Server Platform". In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. 2018, pp. 653–660.

[72] Hosein Mohammadi Makrani, Hossein Sayadi, Sai Manoj Pudukotai Dinakarra, Setareh Rafatirad, and Houman Homayoun. "A Comprehensive Memory Analysis of Data In-

tensive Workloads on Server Class Architecture". In: *Proceedings of the International Symposium on Memory Systems*. 2018, pp. 19–30.

[73] Hosein Mohammadi Makrani, Hossein Sayadi, Sai Manoj, Setareh Raftirad, and Houman Homayoun. "Compressive Sensing on Storage Data: An Effective Solution to Alleviate I/O Bottleneck in Data-Intensive Workloads". In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2018, pp. 1–8.

[74] Hosein Mohammadi Makrani, Hossein Sayadi, Devang Motwani, Han Wang, Setareh Rafatirad, and Houman Homayoun. "Energy-Aware and Machine Learning-Based Resource Provisioning of In-Memory Analytics on Cloud". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2018, pp. 517–517.

[75] Hosein Mohammadi Makrani, Shahab Tabatabaei, Setareh Rafatirad, and Houman Homayoun. "Understanding the Role of Memory Subsystem on Performance and Energy-Efficiency of Hadoop Applications". In: *Green and Sustainable Computing Conference (IGSC), 2017 Eighth International*. IEEE. 2017, pp. 1–6.

[76] Maria Malik, Avesta Sasan, Rajiv Joshi, Setareh Rafatirah, and Houman Homayoun. "Characterizing Hadoop Applications on Microservers for Performance and Energy Efficiency Optimizations". In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2016, pp. 153–154.

[77] *Martin A. Brown. Traffic control howto. http://linux-ip.net/articles/Traffic-Control-HOWTO/.*

[78] Kevin Mills, James Filliben, and Christopher Dabrowski. "Comparing VM-Placement Algorithms for On-Demand Clouds". In: *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. IEEE. 2011, pp. 91–98.

[79] Seyed Ahmad Mirsalari, Najmeh Nazari, Seyed Ali Ansarmohammadi, Sima Sinaei, Mostafa E Salehi, and Masoud Daneshtalab. "ELC-ECG: Efficient LSTM Cell for

ECG Classification based on Quantized Architecture". In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2021, pp. 1–5.

[80] Seyed Ahmad Mirsalari, Najmeh Nazari, Sima Sinaei, Mostafa E Salehi, and Masoud Daneshtalab. "FaCT-LSTM: Fast and Compact Ternary Architecture for LSTM Recurrent Neural Networks". In: *IEEE Design & Test* (2021).

[81] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. "PAPI: A Portable Interface to Hardware Performance Counters". In: *Proceedings of the department of defense HPCMP users group conference*. Vol. 710. 1999.

[82] Senthil Nathan, Umesh Bellur, and Purushottam Kulkarni. "Towards a Comprehensive Performance Model of Virtual Machine Live Migration". In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 2015, pp. 288–301.

[83] Najmeh Nazari, Mohammad Loni, Mostafa E Salehi, Masoud Daneshtalab, and Mikael Sjodin. "TOT-Net: An Endeavor Toward Optimizing Ternary Neural Networks". In: *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE. 2019, pp. 305–312.

[84] Najmeh Nazari, Seyed Ahmad Mirsalari, Sima Sinaei, Mostafa E Salehi, and Masoud Daneshtalab. "Multi-Level Binarized LSTM in EEG Classification for Wearable Devices". In: *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2020, pp. 175–181.

[85] Nameh Nazari and Mostafa E.Salehi. "Hardware Architectures for Deep Learning". In: ed. by Masoud Daneshtalab and Mehdi Modarressi. Materials, Circuits and Device, 2020. Chap. Binary Neural Networks, pp. 95–115.

[86] Katayoun Neshatpour, Hosein Mohammadi Makrani, Avesta Sasan, Hassan Ghasemzadeh, Setareh Rafatirad, and Houman Homayoun. "Design Space Exploration for Hardware Acceleration of Machine Learning Applications in MapReduce". In: *2018 IEEE 26th*

Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE. 2018, pp. 221–221.

[87] Katayoun Neshatpour, Hosein Mohammadi Mokrani, Avesta Sasan, Hassan Ghasemzadeh, Setareh Rafatirad, and Houman Homayoun. “Architectural Considerations for FPGA Acceleration of Machine Learning Applications in MapReduce”. In: *Proceedings of the 18th international conference on embedded computer systems: Architectures, modeling, and simulation*. 2018, pp. 89–96.

[88] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. “Making Sense of Performance in Data Analytics Frameworks”. In: *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 2015, pp. 293–307.

[89] Fengfeng Pan, Yinliang Yue, Jin Xiong, and Daxiang Hao. “I/O Characterization of Big Data Workloads in Datacenters”. In: *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*. Springer. 2014, pp. 85–97.

[90] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. “Practical Black-Box Attacks Against Deep Learning Systems Using Adversarial Examples”. In: *arXiv preprint arXiv:1602.02697* 1.2 (2016), p. 3.

[91] Mathias Payer. “HexPADS: A Platform to Detect “Stealth” Attacks”. In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2016, pp. 138–154.

[92] S Raghavan, P Sarwesh, C Marimuthu, and K Chandrasekaran. “Bat Algorithm for Scheduling Workflow Applications in Cloud”. In: *2015 International Conference on Electronic Design, Computer Networks & Automated Verification (EDCAV)*. IEEE. 2015, pp. 139–144.

[93] Zujie Ren, Xianghua Xu, Jian Wan, Weisong Shi, and Min Zhou. “Workload Characterization on a Production Hadoop Cluster: A Case Study on Taobao”. In: *2012*

IEEE International Symposium on Workload Characterization (IISWC). IEEE. 2012, pp. 3–13.

[94] *Rightscale Inc. 2017. Amazon EC2: Rightscale. http://www.rightscale.com/. (2017).*

[95] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds". In: *Proceedings of the 16th ACM conference on Computer and communications security*. 2009, pp. 199–212.

[96] Francisco Romero and Christina Delimitrou. "Mage: Online Interference-Aware Scheduling in Multi-Scale Heterogeneous Systems". In: (2018), pp. 1–13.

[97] Hossein Sayadi, Yifeng Gao, Hosein Makrani, Avesta Sasan, Jessica Lin, Setareh Rafatirad, and Houman Homayoun. "Towards Run-Time Hardware-Assisted Stealthy Malware Detection". In: (2020).

[98] Hossein Sayadi, Yifeng Gao, Hosein Mohammadi Makrani, Tinoosh Mohsenin, Avesta Sasan, Setareh Rafatirad, Jessica Lin, and Houman Homayoun. "StealthMiner: Specialized Time Series Machine Learning for Run-Time Stealthy Malware Detection based on Microarchitectural Features". In: *Proceedings of the 2020 on Great Lakes Symposium on VLSI*. 2020, pp. 175–180.

[99] Hossein Sayadi, Hosein Mohammadi Makrani, Sai Manoj Pudukotai Dinakarrao, Tinoosh Mohsenin, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. "2SMART: A Two-Stage Machine Learning-Based Approach for Run-Time Specialized Hardware-Assisted Malware Detection". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 728–733.

[100] Hossein Sayadi, Nisarg Patel, Sai Manoj PD, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. "Ensemble Learning for Effective Run-Time Hardware-Based Malware Detection: A Comprehensive Analysis and Classification". In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE. 2018, pp. 1–6.

[101] Hossein Sayadi, Han Wang, Tahereh Miari, Hosein Mohammadi Makrani, Mehrdad Aliasgari, Setareh Rafatirad, and Houman Homayoun. "Recent advancements in microarchitectural security: Review of machine learning countermeasures". In: *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE. 2020, pp. 949–952.

[102] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. "Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 460–471.

[103] Yakun Sophia Shao and David Brooks. "ISA-Independent Workload Characterization and its Implications for Specialized Architectures". In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2013, pp. 245–255.

[104] Selmar K Smit and Agoston E Eiben. "Comparing Parameter Tuning Methods for Evolutionary Algorithms". In: *2009 IEEE Congress on Evolutionary Computation*. IEEE. 2009, pp. 399–406.

[105] Yunsheng Song, Jiye Liang, Jing Lu, and Xingwang Zhao. "An Efficient Instance Selection Algorithm for K Nearest Neighbor Regression". In: *Neurocomputing* 251 (2017), pp. 26–34.

[106] *The Apache Software Foundation, Available at: https://cloudstack.apache.org/*.

[107] *The Apache Software Foundation, Available at: https://hadoop.apache.org/*.

[108] Virginia Torczon and Michael W Trosset. "From Evolutionary Operation to Parallel Direct Search: Pattern Search Algorithms for Numerical Optimization". In: *Computing Science and Statistics* (1998), pp. 396–401.

[109] Theodore B Trafalis and Huseyin Ince. "Support Vector Machine for Regression and Applications to Financial Forecasting". In: *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Comput-*

ing: New Challenges and Perspectives for the New Millennium. Vol. 6. IEEE. 2000, pp. 348–353.

[110] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M Swift. "Resource-Freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense)". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 281–292.

[111] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. "Apache hadoop yarn: Yet another resource negotiator". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. 2013, pp. 1–16.

[112] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics". In: *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 2016, pp. 363–378.

[113] Han Wang, Hossein Sayadi, Gaurav Kolhe, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. "Phased-Guard: Multi-Phase Machine Learning Framework for Detection and Identification of Zero-Day Microarchitectural Side-Channel Attacks". In: *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE. 2020, pp. 648–655.

[114] Han Wang, Hossein Sayadi, Tinoosh Mohsenin, Liang Zhao, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. "Mitigating Cache-Based Side-Channel Attacks through Randomization: A Comprehensive System and Architecture Level Analysis". In: Annual Conference, Exhibition on Design, Automation, and Test in Europe (DATE). 2020.

[115] Han Wang, Hossein Sayadi, Setareh Rafatirad, Avesta Sasan, and Houman Homayoun. "SCARF: Detecting Side-Channel Attacks at Real-Time Using Low-Level Hard-

ware Features”. In: *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE. 2020, pp. 1–6.

[116] Han Wang, Hossein Sayadi, Avesta Sasan, PD Sai Manoj, Setareh Rafatirad, and Houman Homayoun. “Machine Learning-Assisted Website Fingerprinting Attacks with Side-Channel Information: A Comprehensive Analysis and Characterization”. In: *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. IEEE. 2021, pp. 79–84.

[117] Han Wang, Hossein Sayadi, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. “HybriDG: Hybrid Dynamic Time Warping and Gaussian Distribution Model for Detecting Emerging Zero-Day Microarchitectural Side-Channel Attacks”. In: *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2020, pp. 604–611.

[118] Han Wang, Hossein Sayadi, Avesta Sasan, Setareh Rafatirad, Tinoosh Mohsenin, and Houman Homayoun. “Comprehensive Evaluation of Machine Learning Countermeasures for Detecting Microarchitectural Side-Channel Attacks”. In: *Proceedings of the 2020 on Great Lakes Symposium on VLSI*. 2020, pp. 181–186.

[119] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. “Bigdatabench: A Big Data Benchmark Suite from Internet Services”. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2014, pp. 488–499.

[120] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. “SLA-based Admission Control for a Software-as-a-Service Provider in Cloud Computing Environments”. In: *Journal of Computer and System Sciences* 78.5 (2012), pp. 1280–1299.

[121] Zhangjun Wu, Zhiwei Ni, Lichuan Gu, and Xiao Liu. "A Revised Discrete Particle Swarm Optimization for Cloud Workflow Scheduling". In: *2010 International Conference on Computational Intelligence and Security*. IEEE. 2010, pp. 184–188.

[122] Zhenyu Wu, Zhang Xu, and Haining Wang. "Whispers in the Hyper-Space: High-Bandwidth and Reliable Covert Channel Attacks Inside the Cloud". In: *IEEE/ACM Transactions on Networking* 23.2 (2014), pp. 603–615.

[123] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation". In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 19–35.

[124] Wen Xiong, Zhibin Yu, Zhendong Bei, Juanjuan Zhao, Fan Zhang, Yubin Zou, Xue Bai, Ye Li, and Chengzhong Xu. "A Characterization of Big Data Benchmarks". In: *2013 IEEE international conference on big data*. IEEE. 2013, pp. 118–125.

[125] Zhang Xu, Haining Wang, and Zhenyu Wu. "A Measurement Study on Co-Residence Threat Inside the Cloud". In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, pp. 929–944.

[126] Neeraja J Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. "Wrangler: Predictable and Faster Jobs Using Fewer Resources". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2014, pp. 1–14.

[127] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. "Selecting the Best VM Across Multiple Public Clouds: A Data-Driven Performance Modeling Approach". In: *Proceedings of the 2017 Symposium on Cloud Computing*. 2017, pp. 452–465.

[128] Ahmad Yasin. "A Top-Down Method for Performance Analysis and Counters Architecture". In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2014, pp. 35–44.

[129] Tao Ye and Shivkumar Kalyanaraman. "A Recursive Random Search Algorithm for Large-Scale Network Parameter Configuration". In: (2003), pp. 196–205.

[130] Zhibin Yu, Zhendong Bei, and Xuehai Qian. "Datasize-Aware High Dimensional Configurations Auto-Tuning of In-Memory Cluster Computing". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 2018, pp. 564–577.

[131] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012, pp. 15–28.

[132] Seyed Majid Zahedi and Benjamin C Lee. "REF: Resource Elasticity Fairness with Sharing Incentives for Multiprocessors". In: *ACM SIGPLAN Notices* 49.4 (2014), pp. 145–160.

[133] Fucen Zeng, Lin Qiao, Mingliang Liu, and Zhizhong Tang. "Memory Performance Characterization of SPEC CPU2006 Benchmarks using TSIM". In: *Physics Procedia* 33 (2012), pp. 1029–1035.

[134] G Peter Zhang. "Time Series Forecasting Using a Hybrid ARIMA and Neural Network Model". In: *Neurocomputing* 50 (2003), pp. 159–175.

[135] Qi Zhang, Mohamed Faten Zhani, Shuo Zhang, Quanyan Zhu, Raouf Boutaba, and Joseph L Hellerstein. "Dynamic Energy-Aware Capacity Provisioning for Cloud Computing Environments". In: *Proceedings of the 9th International Conference on Autonomic Computing*. 2012, pp. 145–154.

[136] Weijuan Zhang, Xiaoqi Jia, Chang Wang, Shengzhi Zhang, Qingjia Huang, Mingsheng Wang, and Peng Liu. "A Comprehensive Study of Co-Residence Threat in

Multi-Tenant Public PaaS clouds". In: *International Conference on Information and Communications Security*. Springer. 2016, pp. 361–375.

[137] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K Reiter. "Homealone: Co-Residency Detection in the Cloud via Side-Channel Analysis". In: *2011 IEEE symposium on security and privacy*. IEEE. 2011, pp. 313–328.

[138] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. "Cross-Tenant Side-Channel Attacks in PaaS Clouds". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, pp. 990–1003.

[139] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. "Cross-VM Side Channels and Their Use to Extract Private Keys". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM. 2012, pp. 305–316.

[140] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. "BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning". In: *Proceedings of the 2017 Symposium on Cloud Computing*. 2017, pp. 338–350.

[141] Zhichun Zhu and Zhao Zhang. "A Performance Comparison of DRAM Memory System Optimizations for SMT Processors". In: *11th International symposium on high-performance computer architecture*. IEEE. 2005, pp. 213–224.

[142] Paul Zikopoulos, Chris Eaton, and IBM. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 2011.