CoolCloud: improving energy efficiency in virtualized data centers

Zhiming Zhang
Iowa State University
CoolCloud: Improving energy efficiency in virtualized data centers

by

Zhiming Zhang

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
J. Morris Chang, Major Professor
Zhao Zhang
Joseph Zambreno
Phillip H. Jones
Tien N. Nguyen

Iowa State University
Ames, Iowa
2016

Copyright © Zhiming Zhang, 2016. All rights reserved.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ......................................................... vi

ABSTRACT ................................................................. viii

CHAPTER 1. OVERVIEW ................................................... 1

CHAPTER 2. A COOL SCHEDULER FOR MULTI-CORE SYSTEMS EXPLOITING PROGRAM PHASES .................................................. 5

2.1 Introduction .......................................................... 6

2.2 Related Work ........................................................ 9

2.3 Motivation and Model .............................................. 11

2.3.1 Theoretical Bounds of DVFS Energy Savings ............... 13

2.3.2 Model ........................................................... 14

2.3.3 Model Evaluation ............................................... 17

2.4 Feedback Based Voltage and Frequency Scheduler .......... 19

2.4.1 Frequency Mismatch Feedback ................................ 20

2.4.2 Feedback due to DVFS Overhead ................................ 22

2.4.3 Total Feedback .................................................. 23

2.5 Implementation ...................................................... 24

2.5.1 Data Collection for Each Thread ............................ 24

2.5.2 Thread Migration and DVFS Operation .................... 25

2.6 Experiment Result ................................................. 26

2.6.1 Evaluation Result with SPEC CPU2006 .................... 29
CHAPTER 3. COOLCLOUD: A PRACTICAL DYNAMIC VIRTUAL MACHINE PLACEMENT FRAMEWORK FOR ENERGY AWARE DATA CENTERS

3.1 Introduction

3.2 System Model

3.2.1 VM Placement Problem Formulation

3.3 Heuristic Design for Practical Deployment

3.3.1 Algorithm Design Principle

3.3.2 Algorithm Description

3.4 Testbed Implementation

3.5 Experiment Result

3.5.1 Evaluation on Testbed

3.5.2 Evaluation through Simulation

3.6 Related Work

3.7 Conclusion

CHAPTER 4. TAMING ENERGY CONSUMPTION IN DATA CENTERS WITH DYNAMIC VIRTUAL MACHINE PLACEMENT

4.1 Introduction

4.2 Related Work

4.3 System Model

4.3.1 Live Migration

4.3.2 VM Placement Problem Formulation

4.4 Approach to Solve the Problem

4.4.1 Design Principle

4.4.2 Algorithm Details
CHAPTER 4. COOLCLOUD: OPTIMIZING ENERGY EFFICIENCY IN VIRTUALIZED DATA CENTERS

4.5 Testbed and CoolCloud Implementation ........................................... 77
  4.5.1 Testbed Data Center Configuration ........................................... 77
  4.5.2 Live Migration ......................................................................... 79
  4.5.3 CoolCloud Software Design ....................................................... 80
  4.5.4 VMware DRS and DPM ............................................................ 81

4.6 Experiment Result ................................................................. 82
  4.6.1 Evaluation on Testbed ............................................................. 83
  4.6.2 Evaluation through Simulation ................................................ 90

4.7 Conclusion ................................................................. 92

CHAPTER 5. COOLCLOUD: OPTIMIZING ENERGY EFFICIENCY IN VIRTUALIZED DATA CENTERS ................................................................. 94

5.1 Introduction .......................................................... 95

5.2 Design Background ...................................................... 98
  5.2.1 Virtualized Data Center .......................................................... 98
  5.2.2 Dynamic Voltage and Frequency scaling ................................... 101

5.3 CoolCloud Framework Design ........................................ 105
  5.3.1 Local Server Level Optimization ........................................ 105
  5.3.2 MAPI prediction model ......................................................... 108
  5.3.3 Cluster Level Optimization ................................................... 112

5.4 Implementation of CoolCloud ............................................. 119
  5.4.1 Local Server Level ................................................................. 121
  5.4.2 CoolCloud Software Design .................................................. 123

5.5 Experiment Result ....................................................... 124
  5.5.1 Design Overhead and Improvement Breakdown ....................... 129

5.6 Related Work .......................................................... 131
  5.6.1 DVFS Designs ................................................................. 131
  5.6.2 Energy Aware VM Placement .............................................. 132
5.7 Conclusion ................................................................. 133

CHAPTER 6. CONCLUDING REMARKS AND FUTURE WORK ............... 135
Today happens to be Thanksgiving day while I am writing this acknowledgement, a perfect day to say thank you to the people who have supported me all the way during my PhD years. Without these people, this journey would not be possible, let alone this thesis. Thank you all.

I would like to thank my major professor, Dr. Morris Chang for his careful guidance and for all the time he has spent cultivating me. I still remember the time that I flew 13 hours from Beijing and arrived in Des Moines, it was Dr. Chang who picked me up from the airport which warmed my heart as I was missing home so much. His careful attention helped me to quickly get used to this new environment. After starting to do research, I learnt the principles on how to approach research topics, how to identify important problems and how to solve any problem in general which served me well over the years. Dr. Chang is always available to me for discussions, his passion and his energy really motivates me to work hard. He always gives me freedom and encourages me to think of any research that really interests me, that I think is important. Because of this, I started my thesis in energy aware designs in virtualized data centers. There has been many obstacles on the way working towards this thesis, but we were able to overcome all the challenges together.

I would like to thank all my committee members for their time and support of my research. I am grateful for having them on my committee and their inputs that raised the quality of my research. Dr. Zhao Zhang is always nice and kind, he provided me many insights into the architecture details of my local server DVFS design. Dr. Joseph Zambreno is smart and humorous, his questions such as ”what does it actually mean to use EDP” inspires me to think deeper of the evaluation of my design. Dr. Phillip Jones is hardworking and patient, his inputs helped to be more critical on my experiments with the DVFS scheduler. Dr. Tien Nguyen is
knowledgable and responsive, his deep knowledge on software engineering teaches me to look at my research problems from different angles.

I would like to thank my parents, they are the most important people to me. I can always feel their love, their care deep in my heart. They have been the most supportive people. I would like to dedicate this thesis to them. I would also like to thank my brother, as he is working towards his masters degree in finance at Iowa State, I feel very lucky to have a family member here in Ames. I would like to thank my beautiful girlfriend, her love, her patience, her understanding makes me more sensible and enjoy everything more. As a PhD student herself, her excellence in her field also drives me and motivates me to set a high standard for myself. I would also like to thank all my friends, I can share my happiness and stress, because of them, my PhD life is full of hardwork and fun.

Finally, I would like to thank Iowa State University for offering such a beautiful campus and all the nice people. The peaceful environment allows me to stay patient and focused. Throughout the PhD years, I have figured out the goal and purpose of my life and I have acquired the knowledge and will to work toward it. I am prepared and ready for the challenges in the next stage of my life.

In summary, my PhD is a hardworking, humbling and rewarding journey. I will not hesitate to do this again. In the end, I would like to again say thank you to this journey and to the people who have supported me. Happy Thanksgiving.
In recent years, cloud computing services continue to grow and has become more pervasive and indispensable in people’s lives. The energy consumption continues to rise as more and more data centers are being built. How to provide a more energy efficient data center infrastructure that can support today’s cloud computing services has become one of the most important issues in the field of cloud computing research.

In this thesis, we mainly tackle three research problems: 1. how to achieve energy savings in a virtualized data center environment; 2. how to maintain service level agreements; 3. how to make our design practical for actual implementation in enterprise data centers. Combining all the studies above, we propose an optimization framework named CoolCloud to minimize energy consumption in virtualized data centers with the service level agreement taken into consideration. The proposed framework minimizes energy at two different layers: (1) minimize local server energy using dynamic voltage & frequency scaling (DVFS) exploiting runtime program phases. (2) minimize global cluster energy using dynamic mapping between virtual machines (VMs) and servers based on each VM’s resource requirement. Such optimization leads to the most economical way to operate an enterprise data center.

On each local server, we develop a voltage and frequency scheduler that can provide CPU energy savings under applications’ or virtual machines’ specified SLA requirements by exploiting applications’ run-time program phases. At the cluster level, we propose a practical solution for managing the mappings of VMs to physical servers. This framework solves the problem of finding the most energy efficient way (least resource wastage and least power consumption) of placing the VMs considering their resource requirements.
CHAPTER 1. OVERVIEW

In general, improving energy efficiency in a virtualized data center can be achieved from two levels: the cluster level and the local server level. Energy efficient designs at each level of the data center require a comprehensive understanding of the current state of the art architecture of data centers and available energy saving techniques. In the following chapters, we present the challenges of achieving energy savings and maintaining service level agreements at each level which leads to the design of a practical optimization framework: CoolCloud.

In Chapter 2, we discuss the challenges of using dynamic voltage and frequency scaling to achieve CPU energy savings under the specified SLA requirement by exploiting the applications’ run-time program phases. As previous works show high computation complexity: using regression based model or solving an NP-hard MCKP that increase design overhead. We introduce a simple and effective voltage and frequency scheduler (the “cool” scheduler). Our scheduler greatly improves the computation efficiency compared with other recently published works. The computation complexity for our design is O(1) compared to O(N) in previous works. We first construct a simple model (the ”cool” model) to calculate a desired running frequency for each thread given its program phases and SLA requirement. In our SLA model, the SLA defines a task execution time constraint for the CPU. We verify our cool model against the industry standard benchmarks from SPEC CPU2006. Verification result shows our model has accurate prediction on most of the benchmark programs. After the desired operating frequency is determined for each thread, thread migration and task grouping are used to perform DVFS for a group of threads in a multi-core environment. This idea significantly reduces the number of unnecessary DVFS operations in recent works. We propose a feedback mechanism
to ensure the actual performance approach closely to the SLA requirement. This allows our cool scheduler to precisely control the performance loss and maximize energy saving under the given SLA requirement.

In Chapter 3, we address the problem of finding the optimal mappings between virtual machines and physical servers in a virtualized data center environment. Recent studies focus on improving server resource utilizations, meeting power budgets, balancing workloads among servers and reducing any energy related costs. However, extant approaches have either one or several of the following limitations: 1. only consider solving one specific problem or focus on one aspect of optimization, e.g., balancing VMs across servers, eliminating hot spots, minimizing power consumptions, maximizing resource utilizations, etc; 2. the optimization models do not scale to the size of large enterprise data centers; 3. only focus on one or two server resources at the same time, e.g., CPU, memory or network bandwidth. Such solutions may perform well on the server resource(s) being considered and leave potential performance bottlenecks on the resource(s) being left out; 4. only provide the initial placement of VMs without taking care of the runtime workload fluctuations; 5. evaluations are mostly carried out through simulation studies that oversimplifies the real scenarios in data center management; 6. limited workload or benchmark selection that do not well represent today’s cloud computing environment. These limitations hinder recent works from being practically applied to enterprise data centers for higher energy efficiency.

We tackle the above discussed challenges with the goal of designing a practical virtual machine placement framework that can be applied to real world enterprise data centers. We formulate the VM placement problem into an ILP optimization problem with the objective of maximizing cluster energy savings. Due to that the optimization is NP hard, a heuristic approach is further proposed to reduce computation complexity and make our design scale well to the size of enterprise data centers. VM Live migration (with its cost considered) is used to move VMs from one server to another when placement decisions are made. A real testbed data center implemented with industry product VMware vSphere 5 is used to evaluate the proposed
framework. The main contributions of this work includes: 1. The optimization design can achieve maximum energy savings with all resource constraints (CPU, memory, network and storage) and VM live migration costs taken into account. 2. The framework is a practical solution that can be applied to enterprise data centers. The computation efficient heuristic design provides fast placement solutions given workload fluctuations. 3. The design is implemented and evaluated within a real testbed built from industry leading platform. Experiment result suggests that the proposed design can effectively improve data center energy efficiency and is highly scalable for large size data centers.

In Chapter 4, we extend the work of the dynamic virtual machine placement design. A VM placement algorithm with low computation complexity based on quick sort and greedy algorithm is designed to solve the placing problem timely. The efficient algorithm is key to making dynamic VM placement scalable to large size data centers. Maximum energy saving is achieved through consolidating VMs to the least number of servers and turning remaining servers to sleep mode. The resource monitoring process collects server resource utilization of all aspects, which is key to eliminating resource wastage or performance bottlenecks. The algorithm does not only provides the initial placement of VMs, but proactively monitors runtime workload fluctuations and provides new placement solutions in case of service level agreement violations or a more energy efficient placement is discovered. A new placement map will be generated when a new solution is discovered, VMs will be migrated to their designated server location according to the placement map. Live migration is used to move VMs around which minimizes downtime and interruption to users. We build a real testbed data center to evaluate the proposed design. We choose workloads from web service applications, big data benchmarks, i.e., HiBench to Docker software containers that represent today’s cloud computing environment to thoroughly evaluate our work.

In Chapter 5, we combine our research of applying DVFS according to program phases and dynamically placing virtual machines based on their resource requirements, and propose a new optimization framework named CoolCloud for large scale virtualized data centers around a set
of energy conservation opportunities and service/resource constraints. CoolCloud optimizes energy consumption at both levels: (1) local server level: minimize local server energy using dynamic voltage & frequency scaling (DVFS) exploiting runtime program phases. (2) cluster level: minimize global cluster energy using dynamic mapping between virtual machines (VMs) and servers based on each VM’s resource requirement. Such optimization leads to the most economical way to operate an enterprise data center. The contributions of this work includes: 1. To the best of our knowledge, this work is the first to provide energy and performance optimizations at both the cluster level and the local server level. 2. We propose a MAPI prediction model based on time series analysis that can accurately forecast the next value of MAPI which helps to maintain SLA. We implement CoolCloud with Xenserver 6.5 and build a real data center testbed to evaluate our work. We select industry standard cloud computing benchmarks Hibench 2.6 and workloads that represent the most recent cloud computing applications. Experiment result demonstrates that CoolCloud can effectively provide significant energy savings while maintaining service level agreements.

Finally in Chapter 6, we conclude this dissertation with major contributions of my research and outline my future research directions.
CHAPTER 2. A COOL SCHEDULER FOR MULTI-CORE SYSTEMS
EXPLOITING PROGRAM PHASES

This paper is published in *IEEE Transactions on Computers*\(^1\)

Zhiming Zhang\(^2\), J. Morris Chang\(^3\)

Abstract

Rapid growth of cloud computing services have led to creation of large scale enterprise data centers which consume great amounts of energy. Data centers usually have an SLA (Service Level Agreement) between the clients and the service providers which specify the terms and quality of service to be provided. In this paper, we consider a situation in a data center where multiple user applications are executing on a multi-core system and each application may have a specified SLA requirement. We design a voltage and frequency scheduler (the "cool" scheduler) that can be used in enterprise data centers to provide CPU energy saving under the specified SLA requirement by exploiting the applications’ run-time program phases. Our design greatly improves the computation efficiency compared to other recently published works. The scheduler is built into the Linux kernel and evaluated against SPEC CPU2006 and Phoronix Test Suite on a quad-core system. Experiment result demonstrates our cool scheduler achieves 25.8% energy saving on average with 8.7% performance loss under the given SLA

\(^1\)IEEE Transactions on Computers, VOL. 63, NO. 5, MAY 2014 1061
\(^2\)Primary researcher and author.
\(^3\)Author for correspondence.
requirement (10% allowed performance loss). Our design achieves 35.8% and 31.6% more energy saving compared to two of the most advanced related works.

2.1 Introduction

Energy management has now become a key issue for cloud computing service providers, focusing on the reduction of all energy related costs. Energy proportional computing has become a popular solution to provide energy savings among data centers. The basic idea of energy proportional computing is to minimize energy consumption in data centers while meeting the SLA (Service Level Agreement) requirement.

In general, SLA sets the expectations of service such as throughput and transaction response time between the customer and service provider. The transaction response time can be considered as the waiting time for a customer while the task is being processed in the data center. A data center usually has a minimum transaction response time $t$ which is the case when the data center is operating at its maximum capability or generally the highest frequency. Assume the SLA transaction response time is set as $T$ between the customer and the service provider, then the minimum transaction response time $t$ of the data center must be smaller than $T$. Otherwise this data center can not provide service to this customer since the SLA can not be guaranteed. If $t$ is smaller than $T$, there is chance for energy savings since the data center can operate on lower frequency but still meet the SLA. The goal of our work is to maximize energy saving without violating the SLA requirement by adjusting the CPU frequency based on application’s run-time program phases in a multi-core system.

Processor frequency has always been a key metric of system performance and higher frequency generally means better overall system response or throughput. However high operating frequency may also lead to high potential of energy waste especially in data centers since cloud computing services usually contain many I/O and memory transactions. Our research attempts to minimize the energy waste caused by memory-related stall cycles by using the technique of
dynamic voltage frequency scaling (DVFS). DVFS is widely used to provide energy efficient computing. Most modern computers support a simple workload based DVFS. When the system detects heavy workload, it will increase CPU frequency to provide high performance, and in the case of little workload, the system will decrease CPU frequency to save energy.

Moreover, consider the energy waste due to the speed gap between CPU and main memory. One ideal solution is to minimize the CPU frequency every time when the CPU is stalled by main memory access, and then switch back to high frequency after the stall is over. In this case, energy can be saved with no performance degradation. However in practice, CPU will be unavailable for about 50 $\mu$s to 650 $\mu$s [48, 2] during a DVFS operation. This time-span is much larger than the main memory latency which is around 100 nano seconds. Thus DVFS can not be applied every time the processor is stalled by a main memory access.

A practical solution is to exploit the program phases (i.e. memory intensive phase and CPU intensive phase [38], [72]). The memory intensive phase is the time duration when the program has many memory activities. We can turn down the processor frequency during this time period to save energy but still achieve comparable performance. The CPU intensive phase is the time duration when most of the work is done on the CPU. The CPU should run on high frequency during this time period to guarantee performance. We call the memory intensive phase and the CPU intensive phase two distinct ”program phases”. Recent works [33], [10], [11], [47] try to reduce the memory-related energy waste by adjusting CPU frequency according to the program phases. However, all these works require high computation complexity and ignore the DVFS operation overheads which are substantial for heavy loaded data centers. Another major issue is these works are unable to precisely control the performance loss and the SLA may be violated. These major issues impede these works from being practically used in real data centers.

In this paper, we introduce a simple and effective voltage and frequency scheduler (the ”cool” scheduler). We name it the cool scheduler because it has the ability to reduce CPU energy consumption and cool down the CPU. In our SLA model, the SLA defines a task execu-
tion time constraint for the CPU. We assume the system performance is dominated by the CPU without considering the changing latency of I/O devices or network accesses. Our scheduler greatly improves the computation efficiency compared with other recently published works. We first construct a simple model (the "cool" model) to calculate a desired running frequency for each thread given its program phases and SLA requirement. We verify our model against the industry standard benchmarks from SPEC CPU2006. Verification result shows our model has accurate prediction on most of the benchmark programs. After the desired operating frequency is determined for each thread, thread migration and task grouping are used to perform DVFS for a group of threads in a multi-core environment. This idea significantly reduces the number of unnecessary DVFS operations in recent works. We propose a feedback mechanism to ensure the actual performance approach closely to the SLA requirement. This allows our cool scheduler to precisely control the performance loss and maximize energy saving under the given SLA requirement.

The scheduler is built into the Linux 2.6.22.9 kernel. We evaluate our work on a desktop computer with Intel Core 2 Quad 8400 CPU against benchmarks from SPEC CPU2006 [28] and Phoronix Test Suite [69]. Experiment result demonstrates our cool scheduler achieves 25.8% energy saving on average with 8.7% performance loss under the given SLA requirement. It also demonstrates our scheduler achieves 35.8% and 31.6% more energy saving respectively compared to two of the most advanced related works. The main contributions of our work are:

- We propose a cool scheduler that can be used in enterprise data centers to provide CPU energy saving under the specified SLA requirement by exploiting the applications’ runtime program phases.

- The proposed scheduler greatly improves the computation efficiency compared to two of the most advanced related works.
• The proposed scheduler significantly reduces the number of unnecessary DVFS operations which are ignored in recent works.

• The proposed scheduler can precisely control the performance loss and maximize energy saving with the SLA requirement always guaranteed.

The remaining of the paper is organized in the following sequence. Related work is given in Section 2.2. We provide our theoretical intuition and the cool model in Section 2.3. Section 2.4 introduces the feedback based voltage and frequency scheduling mechanism. The implementation of our design is provided in section 2.5. Section 2.6 exhibits the experiment results and Section 2.7 concludes this paper.

2.2 Related Work

A number of works have used DVFS related techniques to provide energy efficient computing, we limit our discussion to the methods that are most relevant to our work. Recent research on DVFS based energy efficient techniques can be classified into at least three groups. The first group of techniques use known task arrival times, workload, and deadlines to implement algorithms at the task level or operating system [37, 56, 68, 30, 91, 43, 76, 27, 26]. Horvath et al. [30] proposed a DVFS policy for multi-tier web server system that can minimize global energy consumption while meeting the multi-stage end-to-end delay constraint. Isci et al. [37] analyzed different policies for chip level power management under a specific power budget. These policies adjust power modes of individual cores targeting at different objectives such as prioritization of cores/benchmarks, balancing power among cores and optimizing system throughput.

The second group of techniques use compiler or application support for performing DVFS [59, 22, 52, 55, 4, 92, 44, 87]. For example, in [92], the authors provide an application level power management by using the knowledge provided by the application to save energy. In [59], the authors use dynamic profiling of branch probability to characterize workload then
use DVFS to maintain power-performance balance. This group of methods need additional code added to the application before it is executed on the system.

The last but not the least group of techniques use program runtime characteristics or statistics to identify the workload of a task. Then estimate and predict the optimal voltage and frequency setting [50, 11, 10, 47, 5, 23, 13, 60, 79, 40, 41, 31]. For example, Kotla et al. [50] use the program runtime information instruction per cycle to decide the running frequency, this method can reduce energy waste caused by memory stalls, however the scheme does not guarantee the SLA requirement. These techniques can be further classified as fine-grained or coarse-grained. Course-grained techniques determine the voltage and frequency setting on a task-by-task basis. Fine-grained techniques adjust the voltage and frequency setting within a task boundary and usually perform better than course-grained techniques.

Choi et al. [11] presents a fine-grained DVFS technique that minimizes energy consumption using workload decomposition which classifies workload as either on-chip or off-chip. The authors propose a regression based model to calculate the optimal running frequency for a program. Chen et al. [10] uses last level cache misses per instruction (MPI) as an indicator of energy consumption. Given the program’s MPI distribution, the corresponding energy consumption and other statistics, the DVFS control problem is formulated into a multiple choice knapsack problem (MCKP) with the goal of minimizing total energy consumption.

However, both works require high computation complexity: using regression based model or solving an NP-hard MCKP. Besides, they ignore the DVFS operation overhead (invoking DVFS at every context switch or every 30ms) which is significant for heavy loaded data centers. Another major issue is the prediction errors in these two works. They are unable to precisely control the performance loss and the SLA requirement may be violated. To overcome these issues, we design a DVFS scheduler that has little computation complexity (O(1) compared to O(N) in [11, 10]). We use the idea of task grouping and thread migration [33] to perform DVFS for a group of threads in a multi-core environment. This significantly reduces
the DVFS operation overheads. We propose a feedback mechanism to precisely control the performance loss and maximize energy saving with SLA always guaranteed.

### 2.3 Motivation and Model

Program run-time behavior can be categorized into two phases: memory-intensive phase and CPU-intensive phase [49]. In the memory intensive phase (frequent last level cache miss), the CPU spends significant amounts of time waiting for memory transactions thus wasting energy. Slowing down the CPU frequency during this time could provide energy savings while still achieve comparable performance. We use a simple experiment to illustrate this idea. We execute \textit{mcf} (a benchmark program from SPEC CPU2006 used for single-depot vehicle scheduling in public mass transportation) on two different frequencies and then examine its program behaviors. Figure 2.1 shows the execution behavior (MAPI vs time) of \textit{mcf} when CPU is running on 1.998 GHz and 2.664 GHz respectively. MAPI is the number of Memory Access Per Instruction which can be used as an indicator of a program’s memory access intensiveness. Observation shows two distinct phases: memory intensive phase ($MAPI > 0.008$) and computation intensive phase ($MAPI < 0.006$). The execution time for memory intensive phases is about the same no matter when the program is running on 1.998 GHz or 2.664 GHz. However, CPU running on 1.998 GHz causes the execution time of computation intensive phases obviously longer than when the CPU is running on 2.664 GHz.

Observations demonstrate performance drops when CPU frequency is reduced. For the same amount of frequency drop, the performance degradation depends on the program phases. This implies performance suffers less degradation at memory intensive phase for the same amount of frequency drop. This motivates us to switch down the frequency during memory intensive phases to save energy without much performance degradation. Another important observation is that the program tends to have similar run-time behaviors [71], [16] within a time span. Usual time spans for similar run-time behaviors are in seconds or tens of seconds.
Figure 2.1: Execution behavior of \textit{mcf} on 1.998 GHz and 2.664 GHz
As we observe from the execution of "mcf" in Figure 2.1, the time span for memory intensive phases is about 7 seconds, and about 20 seconds for CPU intensive phases. This feature is used to predict the program’s future behavior.

### 2.3.1 Theoretical Bounds of DVFS Energy Savings

The experiment above provides the motivation to switch down CPU frequency during memory intensive phases for energy savings. In this section, we provide the theoretical bounds of DVFS energy savings when executing a program under a time constraint. Theoretically, when the program contains only memory access instructions, the program will stay in memory intensive phase throughout the execution. In this case, maximum or upper bound of energy saving can be achieved since the program can be executed on the lowest frequency without performance loss (assume CPU is much faster than memory). On the other hand, when the program has no memory access, the program stays in CPU intensive phase throughout the execution. In this case, the minimum or lower bound of energy saving will be reached. These two bounds are given in the following.

Assume the SLA requirement (time constraint) for program execution is \( T \). Consider a CPU that supports multiple operating frequencies and assume under the same time constraint, executing the program under lower frequency consumes less energy. \( F \) represents the maximum CPU frequency. \( T_F \) and \( P_F \) are the execution time and power consumption respectively when CPU is operating at \( F \). We add a constraint that \( T_F \) is smaller than \( T \) so the time constraint can be guaranteed when CPU operates at maximum frequency. \( f_{min} \) represents the minimum CPU frequency. \( T_{f_{min}} \) and \( P_{f_{min}} \) are the execution time and power consumption when CPU is operating at \( f_{min} \).

When the program contains only memory access instructions, it can be executed on \( f_{min} \) without performance loss and the upper bound of energy saving \( \Delta E_U \) is achieved:

\[
\Delta E_U = P_F T_F - P_{f_{min}} T_{f_{min}}
\]  
(2.1)
When the program has no memory access, the lower bound $\Delta E_L$ is reached, where $f_k$ is the CPU frequency that allows $T_{f_k} = T$, so the time constraint is strictly met. $T_{f_k}$ and $P_{f_k}$ are the execution time and power consumption respectively when CPU is operating at $f_k$. When the program contains both memory and non-memory instructions, the amount of energy saving stays between $\Delta E_L$ and $\Delta E_U$. 

### 2.3.2 Model

The energy saving capability of DVFS strategies depends on the SLA and the amount of memory accesses in a program. CPU frequency must be carefully chosen based on the distribution of the memory accesses. The scheduler must be able to identify program phases and make DVFS decisions at runtime. We propose a model that can provide the desired running frequency based on the SLA and the program phases. This model shows great computation efficiency compared to recent works [11][10] and it can be built into OS kernel for commercial use. The program phases are mainly determined by three statistics captured at run-time using performance monitors: $MAPI, CPI_{exe}, h(f)$. We first give definitions for the behavior statistics we use.

- **MAPI**: Memory Access Per Instruction, to determine the memory access intensiveness of a thread [35].
  \[
  MAPI = \frac{Bus\_Trans\_Mem}{Instr\_Exe}
  \]  
  where $Bus\_Trans\_Mem$ is the number of main memory accesses and $Instr\_Exe$ is the number of instructions executed.

- **CPI_{exe}**: cycle per instruction when CPU pipeline not stalled by memory transactions.

- **IC\((f)\)**: instruction count, total number of instruction executed in one second at CPU frequency $f$. 

\[
\Delta E_L = P_F T_F - P_{f_k} T_{f_k}
\]  

(2.2)
• $\Delta m$: latency of the main memory.

• $h(f)$: number of cycles when CPU is halted while operating at frequency $f$.

• $o(f)$: stall cycles caused by reasons other than memory access while CPU running at $f$.

• $\alpha$: memory latency overlap rate. This factor represents the out-of-order execution before CPU gets stalled by a memory access.

The cycle usage for a CPU operating on frequency $f$ within a second can be expressed as:

$$ f = IC(f) \times CPI_{exe} + \alpha \times MAPI \times IC(f) \times \Delta m \times f + h(f) + o(f) $$

(2.4)

where $IC(f) \times CPI_{exe}$ is the number of cycles while the CPU is not stalled by memory transactions neither halted. $\alpha \times MAPI \times IC(f) \times \Delta m \times f$ is the number of stall cycles due to main memory access. Notice that quantities on both sides of eq. (2.4) are in cycles/sec. $h(f)$ represents the number of cycles when CPU is halted while operating at frequency $f$.

The CPU gets halted when there is no work to be done, the CPU starts running an idle thread (HLT instructions) and enters its idle state. CPU stall happens when the CPU is still executing program instructions but waiting for the operand or data (usually because of the latency of memory) to be available.

In a recently published model [11], the authors ignore the effect of out-of-order execution and memory level parallelism [12] in superscaler processors which leads to prediction errors. In our model we define $\alpha$ to represent this effect and enhance the accuracy of our model. The value of $\alpha$ is determined by the processor issue rate, re-order buffer size and system memory latency. In general, most of the stall cycles are caused by main memory access, thus we can ignore the $o(f)$ (e.g. L1 cache miss and branch miss prediction related stalls) in eq. (2.4) with little impact on the accuracy of eq. (2.4). Eq. (2.4) is rewritten into:

$$ f \approx IC(f) \times CPI_{exe} $$

$$ + \alpha \times MAPI \times IC(f) \times \Delta m \times f + h(f) $$

(2.5)
The instruction count $IC(f)$ can be derived from eq. (2.5). In this performance model, we consider instruction count in a given interval of time as the performance measure of a thread [50]. Thus, the performance loss for CPU running on frequency $f$ compared to CPU running on the highest frequency $F$ can be defined as:

$$\delta = \frac{IC(F) - IC(f)}{IC(F)}$$

(2.7)

When the SLA requirement is given as a percentage of the maximum system performance, the required performance loss can be calculated as:

$$\delta = 1 - SLA$$

(2.8)

Consider a time-sharing multi-tasking system, each thread is given a time slice to execute on the CPU. Let $f_{n-1}$ be the frequency level of a thread $t$’s $(n-1)$th execution, its program behaviors $CPI_{exe}$, $MAPI$, $h(f)$ are monitored by the CPU during $t$’s $(n-1)$th execution. Experiment demonstrates that the number of halted cycles $h(f)$ depends on CPU frequency:

$$h(f_1) \approx \frac{f_1}{f_2} \times h(f_2)$$

(2.9)

where $f_1$ and $f_2$ are two different CPU frequencies. After collecting all the program behavior statistics, combine eq. (2.6), (2.7), (2.9) and obtain eq. (2.10), which is the equation that provides the desired operating frequency for $t$’s $n$th execution:

$$f_{\text{target}}^n = \frac{IC(F) \times (1 - \delta) \times CPI_{exe}}{1 - IC(F)(1 - \delta) \alpha \times MAPI \times \Delta m \times \frac{h(f_{n-1})}{f_{n-1}}}$$

(2.10)

where $IC(F)$ can be derived from eq. (2.6).

$$IC(F) \approx \frac{F - f_{n-1} h(f_{n-1})}{\alpha \times MAPI \times \Delta m \times F + CPI_{exe}}$$

(2.11)

Eq. (2.10) is our proposed model that provides the desired operating frequency $f_{\text{target}}$ for a thread given its program phases and SLA requirement (i.e. target performance loss $\delta$). The computation complexity of our model is O(1) compared to O(N) in two recent works [11][10].
2.3.3 Model Evaluation

The accuracy of our model is evaluated against benchmark programs from SPEC CPU2006. System configuration is given in Section 2.6. The basic evaluation idea is to compare the value derived from the proposed model with the actual value from the performance monitors. First, capture the behavior statistics used in the model while running the benchmark programs on the highest frequency $F$. Second, calculate the performance loss $\delta$ assuming the operating frequency is set to $f$ which is different from $F$. Finally, compare the calculated performance loss with the actual performance loss to get the accuracy of our model. Eq. (2.12) shows how to calculate the error:

\[
\text{error} = \frac{\delta_{\text{prediction}} - \delta_{\text{actual}}}{\delta_{\text{actual}}} \tag{2.12}
\]

Evaluation result is demonstrated in TABLE 2.1. $MAPI$, $CPI_{\text{exe}}$, and the error for each program is shown in the table. The benchmark programs are executed on $F=2.664$ GHz, $f_1=2.333$ GHz, and $f_2=1.998$ GHz each three times. The average performance loss is calculated for comparison. The error rate ranges from 0.5% to 7.8% and 2.1% on average. The result demonstrates our model can make accurate prediction on most of the benchmark programs. The error in our model mainly comes from the inaccurate estimation of the effect of out-of-order execution and memory level parallelism that vary at run-time. Another reason for the error are the stalls caused by, e.g. data dependency, branch miss prediction etc. We assume most of the stalls come from memory transactions and ignore other stalls in the model. $astar$ (a program derived from a portable 2D path-finding library [28]) suffers a 7.8% prediction. This is because for $astar$, most of the stalls are caused by data dependencies and branch miss predictions instead of memory transactions (MAPI is less than 0.001). One more reason for the error comes from the estimation of the halted cycle $h(f)$. The actual value deviates slightly from our estimation eq. (2.9).
Table 2.1: Model Evaluation Result

<table>
<thead>
<tr>
<th>Program</th>
<th>MAPI</th>
<th>$CPI_{exe}$</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>0.0014</td>
<td>0.576</td>
<td>0.8%</td>
</tr>
<tr>
<td>astar</td>
<td>0.0009</td>
<td>0.701</td>
<td>7.8%</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.006</td>
<td>0.553</td>
<td>2.3%</td>
</tr>
<tr>
<td>perlbench</td>
<td>0.0015</td>
<td>0.572</td>
<td>1.4%</td>
</tr>
<tr>
<td>gobmk</td>
<td>0.00141</td>
<td>0.581</td>
<td>3.5%</td>
</tr>
<tr>
<td>hmmer</td>
<td>0.000466</td>
<td>0.472</td>
<td>0.6%</td>
</tr>
<tr>
<td>sjeng</td>
<td>0.002</td>
<td>0.612</td>
<td>2.7%</td>
</tr>
<tr>
<td>libquantum</td>
<td>0.0018</td>
<td>0.588</td>
<td>1.5%</td>
</tr>
<tr>
<td>h264ref</td>
<td>0.0015</td>
<td>0.549</td>
<td>0.7%</td>
</tr>
<tr>
<td>omnetpp</td>
<td>0.0014</td>
<td>0.565</td>
<td>0.7%</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>0.002</td>
<td>0.676</td>
<td>1.5%</td>
</tr>
<tr>
<td>mcf</td>
<td>0.013</td>
<td>0.543</td>
<td>1.3%</td>
</tr>
</tbody>
</table>

Figure 2.2: Scheduler Architecture
2.4 Feedback Based Voltage and Frequency Scheduler

The cool model has been derived and provides the desired running frequency for a thread given the SLA requirement and the program phases. In this section, we introduce our cool scheduler. Figure 2.2 demonstrates the architecture of our scheduler. This is a situation where multiple user applications are running on a multi-core server and each application has a specified SLA requirement. The proposed Cool Scheduler works in four steps:

Step 1: Statistics Collection. At the user application level, each App is given an SLA requirement. At the hardware level, the CPU performance monitors (PMs) keep monitoring the Apps’ Program Behavior.

Step 2: Desired Frequency Calculation. All the statistics collected in Step 1 along with the Frequency Mismatch Feedback will be sent to the Cool Model. The model uses the statistics to calculate the desired operating frequency for each thread.

Step 3: Task Grouping. The Group Frequency Selector groups the threads with the same target frequency onto the same CPU core using thread migration.

Step 4: Apply DVFS to each CPU core. After the task grouping is complete, the group frequency is determined by the Group Frequency Selector and applied to each CPU core accordingly.

The goal of Step 3 is to minimize the overhead caused by DVFS operations. Past works [11, 10] ignore the DVFS operation overhead and apply DVFS to each thread every tens of milliseconds. Typically, the DVFS overhead accounts for 1-3% of the task execution time. However, this is a substantial portion of the overall performance degradation. Assume the DVFS overhead ranges from 150 to 250 microseconds (measured in Section 2.6, and DVFS is applied to each thread every 10s of milliseconds. This implies that if the overall performance degradation is 10%, the overhead of DVFS transitions accounts for up to 25% of the degradation (ideally, the performance degradation should only be caused by slowing down CPU frequencies during memory intensive phases). This is why we claim DVFS operation
overheads are substantial and unnecessary DVFS operations need to be reduced. We use the idea of task grouping and thread migration to group the threads that have the same target frequency onto the same CPU core, and then apply DVFS to this core. This process is done by the Group Frequency Selector. The task grouping method can significantly reduce the number of unnecessary DVFS operations.

In Step 4, the target frequency is applied to each core. However, a challenge is that modern CPUs do not provide continuous frequency levels thus a thread might have to execute on a frequency different from its target frequency. This situation is called a frequency mismatch. When frequency mismatch happens, the actual performance loss for this thread will deviate from its target performance loss and this might lead to SLA violation. We propose a feedback mechanism to guarantee the SLA. It also ensures the actual performance approach closely to the SLA. Through this way, our cool scheduler can precisely control the performance loss and maximize energy saving under the given SLA requirement. Continuous feedbacks are provided to each thread after each execution. There are two types of feedbacks in this mechanism: feedback due to frequency mismatch denoted as $\delta_{freq}$, and feedback due to DVFS overhead denoted as $\delta_{DVFS}$.

### 2.4.1 Frequency Mismatch Feedback

Frequency mismatch happens when a thread is being executed on a frequency different from its target frequency. A continuous feedback $\delta_{freq}$ is provided to each thread in case of frequency mismatch. First, definitions used in this feedback mechanism are given as follows:

- $f_{target}^i(t)$: the $f_{target}$ of the thread in its $i$th execution.
- $\delta_{target}^i(t)$: the target performance loss including feedback of thread $t$ in its $i$th execution.

$$
\delta_{target}^i(t) = \frac{IC^i(F(t)) - IC(f_{target}^i(t))}{IC^i(F(t))}
$$

- $\delta_{overall}(t)$: overall target performance loss for thread $t$ which is preset by a user ($\delta = 1 - SLA$).
• \( \delta^i_{\text{operating}}(t) \): actual performance loss for thread \( t \) in its \( i \)th execution.

\[
\delta^i_{\text{operating}}(t) = \frac{IC(F(t)) - IC(f^i_{\text{operating}}(t))}{IC(F(t))}
\]

(2.14)

• \( IC^i(F(t)) \): Instruction count while thread \( t \) running on \( F \) (highest frequency) in its \( i \)th execution.

• \( f^i_{\text{operating}}(t) \): thread \( t \)'s actual operating frequency which might be different from its \( f_{\text{target}} \).

• \( \Delta IC^i(t) \): Instruction count offset due to the difference between the \( f^i_{\text{target}}(t) \) and the \( f^i_{\text{operating}}(t) \).

\[
\Delta IC^i(t) = IC(f^i_{\text{target}}(t)) - IC(f^i_{\text{operating}}(t))
\]

(2.15)

• \( \delta^i_{\text{freq}}(t) \): system feedback due to frequency mismatch from thread \( t \)'s \( i \)th execution.

When the actual performance loss deviates from the target performance loss during thread \( t \)'s \( i \)th execution, there will be an instruction count offset \( \Delta IC^i(t) \) and this offset should be taken into account to determine the \( f^{i+1}_{\text{target}}(t) \). The number of instructions that must be executed in the \((i+1)\)th execution in order to achieve the overall target performance loss \( \delta_{\text{overall}}(t) \) is:

\[
IC(f^{i+1}_{\text{target}}(t)) = IC(f^{i+1}_{\text{target}}(t)) + \Delta IC^i(t)
\]

(2.16)

\( IC(f^{i+1}_{\text{target}}(t)) \) on the right hand side is the original number of instructions that need to be executed in the \((i+1)\)th execution. \( IC(f^{i+1}_{\text{target}}(t)) \) on the left hand side is the actual number of instructions that need to be executed while taking account of the system feedback \( \Delta IC^i(t) \) due to frequency mismatch. Then use eq. (2.17) to calculate the feedback due to frequency mismatch, which should be added to the \( \delta_{\text{overall}}(t) \) to get the value of \( \delta^i_{\text{target}}(t) \):

\[
\delta^i_{\text{freq}}(t) = \frac{IC(f^i_{\text{target}}(t)) - IC(f^i_{\text{operating}}(t))}{IC(F(t))}
\]

(2.17)

Notice that \( \delta^i_{\text{freq}}(t) \) can also be expressed as the difference between actual performance loss and target performance loss \( \delta^i_{\text{target}}(t) \).

\[
\delta^i_{\text{freq}}(t) = \delta^i_{\text{operating}}(t) - \delta^i_{\text{target}}(t)
\]

(2.18)
2.4.2 Feedback due to DVFS Overhead

To minimize the deviation of a thread’s actual performance loss from its target performance loss, the overhead of DVFS transitions also need to be taken into consideration. During the DVFS operation, the processor becomes unavailable for 10µs to 650µs [2], [48]. For heavy loaded data centers with large number of threads, DVFS overhead can degrade a thread’s performance thus should not be ignored. We introduce the feedback $\delta_{DVFS}$ that takes DVFS overhead into account while calculating the $f_{target}$. To clearly illustrate this feedback idea, we demonstrate how to calculate the feedback due to DVFS overhead in context of the Linux 2.6 task scheduler. We first introduce the active array and the expired array in the Linux 2.6 scheduler. The active array [15] has all the threads with remaining timeslices. The expired array contains all the threads that have exhausted their timeslices but are not terminated yet. Their timeslices will be recalculated for the next execution. When the active array becomes empty, i.e., all the threads have exhausted their timeslices, the two arrays are swapped and the expired array becomes the active array and vice versa.

When a thread is in the active array, its performance is affected by the DVFS overheads of the threads with higher priorities. Assume thread $t$ is given a timeslice $T_{exe}^i(t)$ for its $i$th execution. Assume there have been $N_a$ DVFS operations before $t$ starts its $i$th execution. The CPU unavailable time due to DVFS transition is denoted as $T_{unavailable}$. The first component of the DVFS overhead feedback $\delta_{DVFS,active}^i(t)$ is calculated using eq. (2.19).

$$\delta_{DVFS,active}^i(t) = \frac{N_a(t) \times T_{unavailable}}{T_{exe}^i(t)}$$  (2.19)

On the other hand, when the thread enters the expired array after its $i$th execution, it has to wait for the threads in the active array to finish. During this time, assume there have been $N_e$ DVFS operations and thread $t$ is given a timeslice $T_{exe}^i(t)$. The second component of the DVFS overhead feedback $\delta_{DVFS,expired}^i(t)$ is calculated using eq. (2.20).

$$\delta_{DVFS,expired}^i(t) = \frac{N_e(t) \times T_{unavailable}}{T_{exe}^i(t)}$$  (2.20)
\[ \delta_{DVFS}^i(t) = \delta_{DVFS, active}^i(t) + \delta_{DVFS, expired}^i(t). \] (2.21)

Notice that the feedback due to DVFS overhead \( \delta_{DVFS}^i(t) \) is equal to the first component \( \delta_{DVFS, active}^i(t) \) when the thread remains in the active array after its execution. However, if the thread enters the expired array after its execution, \( \delta_{DVFS}^i(t) \) is equal to the sum of the two components as in eq. (2.21). This is because it has to wait for the remaining threads in the active array to finish before it can be put back to the active array again. And once it enters the active array, it also has to wait for the threads with higher priorities to finish. Algorithm 1 shows how to calculate the feedback component due to DVFS overhead.

**Algorithm 1** The feedback due to DVFS overhead

**Require:** A thread \( t \)

**Ensure:** feedback component due to DVFS switches

1. \( N_a(t) \leftarrow \) Number of DVFS switches during the time thread \( t \) staying in the active array and before its start of execution.
2. \( T_{exe}(t) \leftarrow \) Timeslice thread \( t \) is allocated for execution. \( \delta_{DVFS}^i(t) \leftarrow \frac{N_a(t) \times T_{unavailable}}{T_{exe}(t)} \)
3. if (context switch) then
4. if (thread \( t \) enters the expired array) then
5. \( N_e(t) \leftarrow \) Number of DVFS switches during the time interval of thread \( t \) staying in the expired array.
6. \( T_{exe}(t) \leftarrow \) Timeslice thread \( t \) is allocated for execution.
7. else
8. \( N_e(t) \leftarrow 0. \)
9. \( T_{exe}(t) \leftarrow 0 \)
10. end if
11. end if
12. \( \delta_{DVFS}^i(t) \leftarrow \delta_{DVFS}^i(t) + \frac{N_e(t) \times T_{unavailable}}{T_{exe}(t)} \)

### 2.4.3 Total Feedback

After calculating both the frequency mismatch feedback and the DVFS overhead feedback.

We can get the total feedback from a thread \( t \)'s \( i \)th execution:

\[ \delta^i(t) = \delta_{freq}^i(t) + \delta_{DVFS}^i(t) \] (2.22)
After each execution, add this feedback to thread \( i \)’s next execution \((i + 1)\)th to determine its target performance loss. Then put \( \delta_{\text{target}}^{i+1}(t) \) into the model eq. (2.10) to calculate \( f_{\text{target}}^{i+1}(t) \).

\[
\delta_{\text{target}}^{i+1}(t) = \delta_{\text{overall}}(t) - \delta^i(t)
\]  

(2.23)

2.5 Implementation

The cool scheduler is built into the Linux kernel 2.6.22.9. Most of our modifications are on the task scheduler without interfering its original functions. Figure. 2.2 demonstrates our scheduler architecture. In this section, we demonstrate how to capture each thread’s behavior at runtime, how to do thread migration and finally how to apply DVFS to each CPU core.

2.5.1 Data Collection for Each Thread

Performance monitors (PMs) [40], [41], [31] are used to capture the program behavior metrics in the proposed model eq. (2.10). Intel Core 2 processors have five performance counters per core [35], [36]. Performance monitor one (PM1) and performance monitor two (PM2) are fully programmable. PM1 and PM2 can count 116 and 115 different types of events respectively. The other three counters can each count one fixed type of event (for counter 3: INSTR_RETIRED.ANY, 4: CPU_CLK_UNHALTED.CORE, and 5: CPU_CLK_UNHALTED.REF). In the implementation, performance monitor one (PM1) records the number of memory accesses. Performance monitor two (PM2) records the number of stalled cycles. PM3 and PM4 record the number of instructions executed and the number of unhalted cycles respectively. Parameters in the model are then calculated based on the PM values, e.g., \( \text{MAPI} = \frac{\text{PM1}}{\text{PM3}} \), \( \text{CPI}_{\text{exe}} = \frac{\text{PM4} - \text{PM2}}{\text{PM3}} \). The PMs start their data collection for each thread after every system call \( \text{context\_switch}(\text{previous thread, next thread}) \) [15]. The PMs are reset at the next context switch to collect data for a new thread.
Table 2.2: Mapping of $f_{\text{target}}$ to a Phase and CPU Frequency for Intel Core 2 Quad 8400

<table>
<thead>
<tr>
<th>Phase</th>
<th>$f_{\text{target}}$ Range</th>
<th>Mapped CPU Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0.2,1.6] GHz</td>
<td>1.998 GHz</td>
</tr>
<tr>
<td>2</td>
<td>(2.16,2.50] GHz</td>
<td>2.333 GHz</td>
</tr>
<tr>
<td>3</td>
<td>(2.50,2.66] GHz</td>
<td>2.664 GHz</td>
</tr>
</tbody>
</table>

2.5.2 Thread Migration and DVFS Operation

After $f_{\text{target}}$ is determined by the model, it will be mapped to the closest frequency that is supported by the CPU. This mapping strategy allows the program to be executed closest to the most energy efficient way under the frequency mismatch condition. Table 2.2 shows the mapping of a range of $f_{\text{target}}$ to frequencies that are supported by Intel Core 2 Quad 8400. We give each CPU core a phase number to represent its operating frequency. The total number of phases for a given CPU is determined by the number of different frequencies supported by that CPU. Table 2.2 shows that Intel Core 2 Quad 8400 has 3 phases since it supports 3 different frequencies. The phase number is also given to a thread after frequency mapping. For example, if a thread has $f_{\text{target}}=2.4$ GHz, it will be mapped to the closest 2.333 GHz and given a phase number which is 2 in this case.

In our design, we use thread migration to cluster threads with the same phase and put them into the same group. For example, the threads in Table 2.3 are clustered into three groups based on their phases: \{1,2,3,9\},\{4,5,6,7\},\{8,10\} with the numbers representing thread IDs. These three groups are then put on different CPU cores for execution. By using this method of task grouping, DVFS can be applied to a whole group of threads instead of each single thread. As a result, the unnecessary DVFS operations can be significantly reduced. Thread migration is part of the task grouping mechanism and facilitates the group frequency selection. It is operated every time before the active array and the expired array swaps. The proposed thread migration strategy does not interfere with the original load balancing function. For each migration, the migrator checks the expired array of every CPU core $k$ and considers this group as the source group. The expired array of any other CPU core $l$ is considered as the destination group.
Table 2.3: Thread Frequency Mapping Example

<table>
<thead>
<tr>
<th>Thread ID</th>
<th>$f_{\text{target}}$ (GHz)</th>
<th>Mapped CPU Frequency (GHz)</th>
<th>Phase No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.5</td>
<td>1.998</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
<td>1.998</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2.1</td>
<td>1.998</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2.2</td>
<td>2.333</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2.4</td>
<td>2.333</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2.3</td>
<td>2.333</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2.3</td>
<td>2.333</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>2.6</td>
<td>2.664</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>1.7</td>
<td>1.998</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2.6</td>
<td>2.664</td>
<td>3</td>
</tr>
</tbody>
</table>

A simple bidirectional thread exchange is made between the source group and the destination group. An example is given in Figure 2.3. Assume CPU core 1 is in phase 3 and it contains two threads both in phase 1. Since these threads’ phases are different from the CPU core phase, these two threads in phase 1 will be moved to core 2 which is also in phase 1. On the other hand, the thread in phase 3 on core 2 is moved to core 1. For the migration between core 2 and core 3, the idea is the same. Algorithm 2 demonstrates the thread migration policy.

After the group frequency is determined, it is applied to the corresponding CPU core. For Intel processors, the CPU has a *p-state* to represent a frequency and voltage operating state. CPU frequency is adjusted by writing a corresponding value of *p-state* to the *IA32_PERF_CTL* register, which is one of the Model Specific Registers (MSRs) [35], [36]. After writing the *p-state*, the CPU will be unavailable for a short period of time due to voltage transitions. The CPU starts operating on the new frequency when the transition completes.

### 2.6 Experiment Result

The proposed scheduler is evaluated on a desktop computer with Intel Core 2 Q8400 processor, FSB 1333 MHz and 4 GB DDR2 memory. Table 2.4 shows the available frequency and voltage levels for Intel Core 2 Quad Q8400 Processor. Benchmark programs used for evaluation are from SPEC CPU2006 and Phoronix Test Suite. We evaluate the scheduler in
Algorithm 2 The Thread Migration Strategy

Require: The expired array $E_k$ of each core $k$

Ensure: Move threads to the group where their phase is the same with the group phase.

1. for each possible CPU core $k$ do
2. $f_{\text{group}}(k)$ ← the group frequency of core $k$
3. for each other CPU core $l$ do
4. $f_{\text{group}}(l)$ ← the group frequency of CPU $l$
5. $S \leftarrow \{a \text{ where } a \in E_k \& f(a) = f_{\text{group}}(l)\}$
6. $D \leftarrow \{b \text{ where } b \in E_l \& f(b) = f_{\text{group}}(k)\}$
7. while $S \neq \emptyset$ & $D \neq \emptyset$ do
8. $p \leftarrow s$ where $s \in S$
9. $q \leftarrow d$ where $d \in D$
10. if $p$ is not movable then
11. $S \leftarrow S - \{p\}$
12. end if
13. if $q$ is not movable then
14. $D \leftarrow D - \{q\}$
15. end if
16. if $p$ and $q$ are movable then
17. $D \leftarrow D - \{q\}$
18. $E_k \leftarrow E_k + \{q\}$
19. $S \leftarrow S - \{p\}$
20. $E_l \leftarrow E_l + \{p\}$
21. end if
22. end while
23. end for
24. end for
the following categories: (i) Performance (SLA requirement). (ii) Energy Consumption. (iii) Energy Delay Product (EDP). (iv) No. of DVFS operations. LMbench [70] is used to measure the main memory latency and the time overhead of the DVFS operation in our system. Measurement results show the main memory latency is 160 cycles and the time overhead for each DVFS operation ranges from 150 to 250 µs. The memory latency overlap $\alpha$ is set to 0.8 in our system configuration. This is because for the CPU we use in our experiment (Intel Core 2 Q8400), the instruction issue rate is 4 with a 128 entry ROB. It takes 32 cycles for the reorder buffer to be full and stall the CPU pipeline. In our system configuration, the memory latency is 160 cycles and when a memory access happens, the CPU can keep executing for 32 cycles before the ROB gets full. Thus 32/160=20% of the memory latency is actually hidden by the out of order execution and $\alpha$ is set to 1-0.2=0.8.

How to measure the CPU energy consumption (including dynamic and leakage) is a challenge given that the CPU power is always changing due to DVFS control. Our approach is to use a current clamp (Fluke i30 [10, 20]) on the 12 V CPU power supply cables. The out-
Table 2.4: Supported Frequency and Voltage for Intel Core 2 Quad 8400 Processor

<table>
<thead>
<tr>
<th>Level</th>
<th>frequency</th>
<th>voltage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.664 GHz</td>
<td>1.288 V</td>
</tr>
<tr>
<td>2</td>
<td>2.333 GHz</td>
<td>1.175 V</td>
</tr>
<tr>
<td>3</td>
<td>1.998 GHz</td>
<td>1.080 V</td>
</tr>
</tbody>
</table>

Output of the current clamp is sampled every 10 ms using an Agilent 34410A digital multimeter [3]. Then we download the data from the multimeter and calculate the total CPU energy consumption. In this experiment, we compare our design with two past works [11, 10], where the allowed performance loss is 10%. Thus we set the target performance loss $\delta$ to 10%. We assume the system performance is dominated by the CPU without considering the changing latency of I/O devices or network accesses. The goal of this experiment is to demonstrate that our cool scheduler can provide the most energy savings and best system efficiency under a given SLA requirement. Our design is compared with three other configurations: (1) the system always operating at the highest frequency 2.664 GHz, (2) The DVFS policy from Choi [11]. and (3) the DVFS policy from Chen [10]. The DVFS policies from Choi and Chen are two of the most advanced related works. In [11], Choi proposed a fine-grained DVFS policy that classifies workload as either on-chip or off-chip. It uses a regression model to calculate the optimal running frequency for a program. In [10], Chen formulated the DVFS problem into a multiple-choice knapsack problem (MCKP). It also exploits the program’s runtime information and periodically solves the MCKP to provide DVFS control.

2.6.1 Evaluation Result with SPEC CPU2006

Figure 2.4 demonstrates the experiment result for the four system configurations. Highest Frequency represents the system always operating at highest frequency. Choi represents the DVFS policy from Choi [11]. Chen represents the DVFS policy from Chen [10]. Cool Scheduler represents our proposed DVFS policy. The results of performance, energy consumption and EDP from Choi, Chen and Cool Scheduler are normalized to the results from the Highest.
Figure 2.4: Results for SPEC CPU2006 Benchmarks ($\delta=10\%$): (a) Performance. (b) Energy Consumption. (c) Energy Delay Product (EDP). (d) No. of DVFS switches.
Frequency. For the the number of DVFS operations, all the results are normalized to Choi since it uses per thread DVFS which invokes DVFS at every context switch.

Figure 2.4 (a) demonstrates the performance for each system configuration. We assume the 100% performance is achieved when the CPU is operating on the highest available frequency. Thus Highest Frequency always has zero performance loss. The performance of other configurations are normalized to Highest Frequency. Choi shows 12.8% performance loss on average with the highest performance loss 17.0% on omnetpp and the lowest performance loss 7.9% on hminer. Chen shows 9.3% performance loss on average with the highest performance loss 13.2% on gobmk and the lowest performance loss 5.4% on libquatum. Notice that there are SLA (δ=10%) violations in both Choi and Chen’s work. This is because of the prediction errors in their models and they do not have a mechanism to guarantee the SLA requirement. The heavy computation overhead and frequent DVFS operations also degrade the actual performance and compromise their energy saving capabilities. The performance loss for each benchmark program shows great deviation from the SLA requirement (ranging from 5.4% to 17.0%). Our proposed cool scheduler shows 8.7% performance loss on average with the highest performance loss 9.7% on xalancbmk and lowest performance loss 6.1% on sjeng. This result proves our scheduler can successfully guarantee the SLA requirement. It also shows the actual performance loss (8.7%) can approach closely to the target performance loss (10%). This allows our proposed scheduler to precisely control the performance loss and maximize energy saving under the given SLA requirement.

Figure 2.4 (b) demonstrates the normalized energy consumption for each configuration. Choi achieves 19.0% energy saving on average with the most energy saving 23.5% on mcf and the least energy saving 11.8% on omnetpp. Chen achieves 19.7% energy saving on average with the most energy saving 32.3% on mcf and the least energy saving 12.4% on xalancbmk. Our cool scheduler achieves 25.8% energy saving on average with the most energy saving 34.5% on mcf and the least energy saving 12.8% on xalancbmk. The result shows our scheduler achieves 35.8% and 31.6% more energy saving compared to Choi and Chen respectively.
The heavy computation overhead and frequent DVFS operations significantly degrade the performance and compromise the energy saving capabilities in Choi and Chen’s work. This result proves our scheduler can provide the most energy saving under the given SLA requirement compared to two of the most advanced related works. All three configurations achieve highest energy saving on mcf, this is because mcf has the most L2 cache misses thus providing the most energy saving opportunities among all benchmark programs.

Figure 2.4 (c) demonstrates the normalized energy delay product (EDP) for each configuration. EDP measures the overall performance and system efficiency. The result shows all three configurations (Choi, Chen, Cool Scheduler) can improve the system efficiency compared to Highest Frequency. The average normalized EDP for Choi and Chen are 93.1% and 88.7% respectively. Our Cool Scheduler has the lowest average normalized EDP 80.9%. This result proves our scheduler has the best system efficiency among all the configurations.

Figure 2.4 (d) demonstrates the normalized number of DVFS operations invoked in each configuration. This number is zero for Highest Frequency. Choi calculates the optimal frequency and invokes DVFS at every context switch. Chen solves the MCKP problem every second and invokes DVFS every 30 ms. It shows a slightly increment in the total number of DVFS operations compared to Choi. Our proposed scheduler uses task grouping to group the threads in the same phase onto the same CPU core, and then invoke DVFS to this group instead of each single thread. The result shows our scheduler reduce the number of DVFS operations by 24 times compared to Choi. This proves our scheduler can significantly reduce the unnecessary DVFS operations in related works.

2.6.2 Evaluation Result with Phoronix Test Suite

The proposed scheduler is further evaluated with multi-threaded benchmarks provided by Phoronix Test Suite [69]. The benchmarks include: Apache, SQLite, 7-Zip, FFmpeg, Stream, Java 2D and PHP. The results are demonstrated in Figure 2.5. We still compare our design with two past works [11, 10]. Figure 2.5(a) demonstrates the normalized performance in
Figure 2.5: Results for Phoronix Test Suite ($\delta=10\%$): (a) Performance. (b) Energy Consumption. (c) Energy Delay Product (EDP). (d) No. of DVFS switches.
each configuration. Choi has a performance loss ranging from 6.6% to 19.2% and 12.7% on average. Chen has a performance loss ranging from 7.2% to 14.0% and 10.4% on average. Both Choi and Chen have shown SLA violations. Our proposed scheduler has a performance loss ranging from 7.3% to 9.2% and 8.5% on average. This result further demonstrates our scheduler can successfully guarantee the SLA requirement and keep the actual performance loss very close to the target performance loss (10%).

Figure. 2.5 (b) demonstrates the normalized energy consumption for each configuration. Choi achieves energy savings ranging from 12.0% to 26.7% and 17.8% on average. Chen achieves energy savings ranging from 12.9% to 24.5% and 18.4% on average. Our scheduler achieves energy savings ranging from 20.4% to 38.8% and 26.5% on average. Results show our scheduler achieves 44.0% and 48.9% more energy savings than Choi and Chen respectively. Figure. 2.5 (c) demonstrates the normalized energy delay product (EDP). Compared to Highest Frequency, Choi has 94.2% and Chen has 91.1% on average. Our scheduler achieves the lowest value: 80.4% on average. This result further proves our scheduler achieves the best system efficiency among all the configurations. Figure. 2.5 (d) demonstrates our scheduler can significantly reduce the number of unnecessary DVFS switches (by more than 30 times) compared to related works.

2.6.3 Design Overhead and Improvement Breakdown

A major improvement of our design is the reduction of system overhead: computation complexity and unnecessary DVFS transitions. Past works have been using regression based model or by solving NP-hard multiple choice knapsack problem (MCKP) to determine the optimal operating frequency. Our implementation demonstrates solving the regression model and MCKP problem within the OS could extend the program execution time by 3% to 7%. Past works also chose to apply DVFS to each thread every tens of milliseconds. These DVFS overheads further accounts for 1-3% of the total program execution time. In summary, our implementation demonstrates past works has 6-10% of overall system overhead. This implies
that if the actual performance degradation for program execution is 10%, over 60% of the
degradation is caused by system overhead which demonstrates significant system inefficiency.
In our design, the computation overhead accounts for less than 0.4% of the program execution
time. Our design uses thread migration and task grouping to reduce the number of DVFS tran-
sitions. Experiment demonstrates the thread migration overhead ranges from tens to hundreds
of micro-seconds. This overhead is negligible compared to the number of unnecessary DVFS
transitions we have reduced. With the cost of model computation, system feedback and thread
migration all taken into consideration, our design only adds a total of 1.3% overhead to the
program execution.

Unlike the two related works, our scheduler can always guarantee the SLA requirement
because of the continuous system feedback. Our scheduler can also significantly reduce the
number of unnecessary DVFS switches because of our thread migration and task grouping
strategy. In this experiment, we turn off the system feedback, thread migration and task group-
ing functions in our design to further evaluate our scheduler against benchmarks from Phoronix
Test Suite. DVFS decisions are made for each thread at every context switch. We compare this
new scheduler with Choi, Chen and our original scheduler. Results demonstrate this new
configuration achieves 9.5% performance loss on average. It achieves 23.1% energy savings
on average, which is still 29.8% and 25.5% more than Choi and Chen respectively. However,
without thread migration and task grouping, this configuration has about the same amount of
DVFS switches as Choi and Chen. Moreover, it has 14.7% less energy savings compared to
our original scheduler. Most importantly, without the continuous feedback, the scheduler is
unable to guarantee the SLA requirement and performance loss ranges from 5.8% to 12.6%.

To summarize, the experiment result demonstrates our proposed voltage and frequency
scheduler provides the most energy saving and the best system efficiency under the given SLA
requirement compared to two of the most advanced related works. The success is due to three
categories of improvement: (1) our scheduler greatly improves the computation efficiency
(O(1) compared to O(N) in Choi and Chen). (2) Task grouping significantly reduces the
number of unnecessary DVFS operations. (3) The feedback mechanism allows the actual performance loss to approach closely to the target performance loss, thus maximizing energy saving opportunities under the SLA requirement.

2.7 Conclusion

This paper presents a voltage and frequency scheduler that can be used in enterprise data centers to provide CPU energy saving under the SLA requirement. The scheduler dynamically adjusts the CPU voltage and frequency level exploiting the run-time program phases. Our design demonstrates significant reduction on the computation overhead and the number of unnecessary DVFS transitions compared to two recently published works. The scheduler is built into the Linux 2.6.22.9 kernel and evaluated against benchmark programs from SPEC CPU2006 and Phoronix Test Suite. Experiment result demonstrates our cool scheduler achieves 25.8% energy saving on average with 8.7% performance loss under the given SLA requirement (10%).
CHAPTER 3. COOLCLOUD: A PRACTICAL DYNAMIC VIRTUAL MACHINE PLACEMENT FRAMEWORK FOR ENERGY AWARE DATA CENTERS

This paper is published in IEEE CLOUD 2015¹
Zhiming Zhang², Chan-Ching Hsu and J. Morris Chang³

abstract

With the continuing growth of cloud computing services, power consumption has become one of the most challenging issues in data center environments. With the support of today’s virtualization technology, the efficiency and flexibility of data center management can be greatly enhanced, creating great energy saving opportunities. However, effective energy aware design is a non-trivial task, considering the size of the data center, the dynamic fluctuation of workloads and the variation of computing resource requests. In this paper, we propose CoolCloud: a practical solution for managing the mappings of VMs to physical servers. This framework solves the problem of finding the most energy efficient way (least resource wastage and least power consumption) of placing the VMs considering their resource requirements. Experiment result demonstrates our design can effectively improve data center energy efficiency and scales well to large size data centers. Comparing with industry leading product VMware’s Distributed Resource Scheduler (DRS), our design offers better performance in both load balancing and power consumption.

¹8th IEEE International Conference on Cloud Computing, June 27-July, 2015, New York, USA
²Primary researcher and author.
³Author for correspondence.
3.1 Introduction

Virtualized data center environment provides a shared hosting infrastructure to customers who need server resources to run their applications. All the applications run inside virtual machines (VMs) which are provisioned and managed on-demand. VMs’ resource utilization (CPU, memory, network, etc.) must be constantly monitored and the data center manager must respond to the changing on-demand resource requests from applications and determine which physical server a VM should be placed on. This is a time consuming task that can not be performed by human operators in a timely fashion considering the complexity and size of the data center.

Virtualized data center management has brought a great amount of research interest. Most of the extant approaches [89, 80, 25, 21] only consider solving one specific problem or focus on one aspect of optimization, e.g., balancing VMs across servers, eliminating hot spots, minimizing power consumptions, maximizing resource utilizations, etc. However, these goals or optimizations should be considered together to build a well-performing data center. Note that some of the objectives may conflict with each other when not handled carefully, making the optimization problem more complicated. Another issue is that past work only focuses on one or two server resources at the same time, e.g., CPU, memory or network bandwidth. These solutions usually perform well on the server resource(s) being considered and leave potential performance bottlenecks on the resource(s) left out.

In this paper, we tackle the above discussed challenges with the goal of designing a practical virtual machine placement framework that can be applied to real world enterprise data centers. We name our design CoolCloud given its capability of cooling down the data center and providing a more energy efficient cloud. We formulate the VM placement problem into an ILP optimization problem with the objective of maximizing cluster energy savings. Due to that the optimization is NP hard, a heuristic approach is further proposed to reduce computation complexity and make our design scale well to the size of enterprise data centers. VM Live
migration (with its cost considered) is used to move VMs from one server to another when placement decisions are made. A real testbed data center implemented with industry product VMware vSphere 5 is used to evaluate the proposed framework. The main contributions of our work are:

- Our optimization design can achieve maximum energy savings with all resource constraints (CPU, memory, network and storage) and VM live migration costs taken into account.

- Our framework is a practical solution that can be applied to enterprise data centers. The computation efficient heuristic design provides fast placement solutions given workload fluctuations.

- Our design is implemented and evaluated within a real testbed built from industry leading platform. Experiment result suggests that CoolCloud can effectively improve data center energy efficiency and is highly scalable for large size data centers.

The remaining of the paper is organized in the following sequence. We provide our VM placement optimization model in Section 3.2. Section 3.3 introduces the heuristic design. The implementation of our design is provided in section 3.4. Section 3.5 demonstrates the experiment results. Related work is given in Section 3.6 and Section 3.7 concludes this paper.

### 3.2 System Model

The proposed CoolCloud design in Figure 3.1 includes three major components. The first component is responsible of collecting runtime resource utilizations of each VM. These resources include CPU, memory, network and storage. The second component is an integer linear programming optimization model (a heuristic approach is proposed later for practical deployment) that provides the optimal VM placement solution. The objective function of this model is to minimize data center energy consumption without affecting each VM’s per-
formance. The model takes each VM’s resource requirements as its constraints to guarantee performance. The third component is a commander responsible of sending out VM migration commands based on the placement solution from the optimization model.

3.2.1 VM Placement Problem Formulation

The following sections presents the optimization model for the virtual machine replacement problem. The problem is to minimize the system energy consumption, denoted as $\mathcal{E}$, by deploying VMs in active mode on physical machines (PMs) with the consideration of the migration cost. The output includes the virtual machine destinations, migration indicators and operation mode of physical machines specified as, $l_{mn}$, $g_{mn}$, and $o_n$. 

Figure 3.1: VM Placement Framework (CoolCloud)
Table 3.1: Definitions of Important Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Number of physical machines to serve virtual machines</td>
</tr>
<tr>
<td>$M$</td>
<td>Number of virtual machines</td>
</tr>
<tr>
<td>$P_{active}$</td>
<td>Basic power level of physical machines in active mode</td>
</tr>
<tr>
<td>$P_{sleep}$</td>
<td>Power level of physical machines in sleep mode</td>
</tr>
<tr>
<td>$Period$</td>
<td>Time period for which the solution pertains</td>
</tr>
<tr>
<td>$P_{migrate_{mn}}$</td>
<td>Power level for VM $m$ migrating to PM $n$</td>
</tr>
<tr>
<td>$T_{migrate_{mn}}$</td>
<td>Time for VM $m$ migrating to PM $n$</td>
</tr>
<tr>
<td>$H_{CPU}$</td>
<td>Limit on CPU utilization of physical machines</td>
</tr>
<tr>
<td>$H_{MEM}$</td>
<td>Limit on memory utilization of physical machines</td>
</tr>
<tr>
<td>$H_{HD}$</td>
<td>Limit on hard disk utilization of physical machines</td>
</tr>
<tr>
<td>$H_{BW}$</td>
<td>Limit on network bandwidth utilization of physical machines</td>
</tr>
<tr>
<td>$N_{VM}$</td>
<td>Set of VMs, $</td>
</tr>
<tr>
<td>$N_{PM}$</td>
<td>Set of PMs, $</td>
</tr>
<tr>
<td>$U_{CPU}$</td>
<td>Virtual machine CPU utilization, $U_{CPU} = {U_{CPU}^m, \forall m \in N_{VM}}$</td>
</tr>
<tr>
<td>$U_{MEM}$</td>
<td>Virtual machine memory utilization, $U_{MEM} = {U_{MEM}^m, \forall m \in N_{VM}}$</td>
</tr>
<tr>
<td>$U_{HD}$</td>
<td>Virtual machine hard disk utilization, $U_{HD} = {U_{HD}^m, \forall m \in N_{VM}}$</td>
</tr>
<tr>
<td>$U_{BW}$</td>
<td>Virtual machine network bandwidth utilization, $U_{BW} = {U_{BW}^m, \forall m \in N_{VM}}$</td>
</tr>
<tr>
<td>$E$</td>
<td>Total system energy consumed</td>
</tr>
<tr>
<td>$\mathcal{L}$</td>
<td>Placement matrix (decision variable), $\mathcal{L} = (l_{mn})_{M \times N}$</td>
</tr>
<tr>
<td>$\mathcal{G}$</td>
<td>Migration matrix (decision variable), $\mathcal{G} = (g_{mn})_{M \times N}$</td>
</tr>
<tr>
<td>$\mathcal{O}$</td>
<td>Operation mode vector (decision variable), $\mathcal{O} = (o_n)_{1 \times N}$</td>
</tr>
</tbody>
</table>

### 3.2.1.1 Decision Variables

The decision variables of the placement problem are presented by two matrices $\mathcal{L}, \mathcal{G}$ and a vector $\mathcal{O}$. $\mathcal{L} = (l_{mn})_{M \times N}$ is the virtual-physical machine incidence matrix, or the *placement matrix* and $\mathcal{G} = (g_{mn})_{M \times N}$ is the virtual machine migrating incidence matrix, or the *migration matrix*; $\mathcal{O} = (o_n)_{1 \times N}$ is the physical machine activation incidence vector, or the *operation mode vector*. The value of a decision variable is determined as follows.

$$l_{mn} = \begin{cases} 
1, \text{ VM } m \text{ is placed on PM } n, \\
\forall m \in N_{VM}, n \in N_{PM}; \\
0, \text{ otherwise,}
\end{cases}$$
When \( l_{mn} \) is asserted, the physical machine \( n \) serves the virtual machine \( m \), provisioning with the power, \( P_{mn} \); \( o_m \) has to set a value of one correspondingly. If VM \( m \) does migrate to PM \( n \), the migration cost involves power, \( P_{migrate}^{mn} \) and time, \( T_{migrate}^{mn} \) for the procedure. It is practical to assume that a physical machine operating actively (\( o_m = 1 \)) consumes energy at a much higher level than in sleep mode.

### 3.2.1.2 Placement Constraint

Each virtual machine can be served by only one physical machine, and it must be placed on one of the physical machines to have the resource granted toward it. The following constraint essentially set up this condition to satisfy.

\[
\sum_{n \in \mathbb{N}_{PM}} l_{mn} = 1, \forall m \in \mathbb{N}_{VM}, \tag{3.1}
\]

Live virtual machine migration is put into action if a virtual machine is decided to be placed on a physical machine \( n \) different from the one it is currently residing on before the optimal solution is provided. Namely, for a machine \( m \), its next placement, \( l_{mn} = 1 \) and its current placement, \( l'_{mn} = 0 \), which is given information initially, are compared to represent that the virtual machine \( m \) is migrated to the physical machine \( n \) from other physical machine. Constraint (3.2) gives the value of \( g_{mn} \) by the comparison of the two states, \( l_{mn} \) and \( l'_{mn} \).

\[
l_{mn} - l_{mn} \cdot l'_{mn} = g_{mn}, \forall m \in \mathbb{N}_{VM}, n \in \mathbb{N}_{PM}, \tag{3.2}
\]
3.2.1.3 Resource Constraints

The service level agreements are categorized by four aspects: CPU, memory, hard disk and network bandwidth utilization. Essentially, the deployed resources of a physical machine cannot exceed a specified utilization level. In this paper, we assume if the resource constraint of each VM can be satisfied, the applications’ SLA can be satisfied as well. The constraints are as follows, where $U_m$ indicates the utilization of virtual machine $m$:

$$\sum_{m \in \mathbb{N}_{VM}} (l_{mn} \cdot U_m^{CPU}) \leq H_n^{CPU}, \forall n \in \mathbb{N}_{PM}, \quad (3.3)$$

$$\sum_{m \in \mathbb{N}_{VM}} (l_{mn} \cdot U_m^{MEM}) \leq H_n^{MEM}, \forall n \in \mathbb{N}_{PM}, \quad (3.4)$$

$$\sum_{m \in \mathbb{N}_{VM}} (l_{mn} \cdot U_m^{HD}) \leq H_n^{HD}, \forall n \in \mathbb{N}_{PM}, \quad (3.5)$$

$$\sum_{m \in \mathbb{N}_{VM}} (l_{mn} \cdot U_m^{BW}) \leq H_n^{BW}, \forall n \in \mathbb{N}_{PM}, \quad (3.6)$$

For physical machines, the resource utilization level is limited to be not over a specific value, for example, $H^{CPU}$ is the maximum CPU utilization level, which could be a measurement of percentage. A set-up percentage less than 100% leaves the margin for new arrival tasks (25% headroom is given in our design). Energy can be consumed at diverse power levels and the execution time are different from virtual machines.

3.2.1.4 Operation Mode Constraints

The following two constraints define in which mode a physical machine will run. If it is an physical machine operating in active mode, $o_m$ is equal to 1 for PM $m$.

$$o_n \leq \sum_{m \in \mathbb{N}_{VM}} l_{mn}, \forall n \in \mathbb{N}_{PM}, \quad (3.7)$$

$$l_{mn} \leq o_n, \forall m \in \mathbb{N}_{VM}, n \in \mathbb{N}_{PM}, \quad (3.8)$$

Constraint (3.7) and (3.8) together satisfy the conditions that a physical machine operates in active mode if and only if it needs to host an active VM. Otherwise, the physical machine will be turned into sleep mode.
3.2.1.5 Objective Function

We define the energy consumption as a summation of the virtual machine execution energy, migration energy, active physical machine energy and sleeping physical machine energy, which are termed below respectively. The next expression presents the energy consumed as a whole, where $\text{Period}$ is the predefined execution time period.

$$
\min \mathcal{E} = \sum_{m \in \text{VM}, n \in \text{PM}} (P_{mn} \cdot \text{Period} \cdot l_{mn}) + \sum_{m \in \text{VM}, n \in \text{PM}} (P_{migrate} \cdot T_{migrate} \cdot g_{mn}) + \sum_{n \in \text{PM}} (P_{active} \cdot \text{Period} \cdot o_n) + \sum_{n \in \text{PM}} [P_{sleep} \cdot \text{Period} \cdot (1 - o_n)].
$$

Eq. (3.9) is the objective function optimizing the total energy consumption of the data center. $P_{mn}$ denotes the required power level of the virtual machine $m$ to operate on the physical machine $n$.

3.3 Heuristic Design for Practical Deployment

The proposed ILP design provides optimal VM placement solutions, however it is NP hard and unpractical for large size data centers. We develop a heuristic approach which solves the formulated problem to avoid the exponential growth in the computation time. The devised algorithm has the goal of offering suboptimal solutions and low computational complexity.

The pseudocode is shown in Algorithm 3, given the same input with what the model will take into account, and is implemented using Java programming language. The output includes the placement of virtual machines, $l_{mn}$, virtual machine migration, $g_{mn}$ and operation mode of physical machines, $o_n$. 
3.3.1 Algorithm Design Principle

The initial placement is taken by the algorithm as a preliminary solution to improve upon. The algorithm first looks for a solution which does not violate the resource constraints. Because of the substantial gap between the operation energy usages of active and sleep modes of a physical machine, turning physical machines to sleep mode when possible can save energy. The attempt is to seek a new solution with an improved energy consumption value by consolidating virtual machines to less physical machines so that physical machines that are originally active can be switched into sleep mode. The algorithm is devised to solve the problem in a timely manner so that suboptimal solutions can be reached to respond to the granularity of fluctuation of the workload. Overall, the computation complexity of the algorithm is $O(MN(\log M)^2 \log N)$ in the worst case.

3.3.2 Algorithm Description

We divide the heuristic into two working stages, feasible solution initialization and virtual machine consolidation. At the first stage, the algorithm works on looking for feasible solution where all the constraints are satisfied. In the case of no constraint violations, the algorithm proceed to the next stage of the heuristic method, taking the initial placement as a feasible solution; or otherwise the initial placement causes one or more constraint violations, in which case, the initial placement is not viable as a feasible solution.

Making the attempt to find an effective solution, the heuristic essentially migrates around VMs onto different physical machines with the fundamental principle of reducing respective resource utilization by firstly, moving VMs requiring large amount of respective resource to another physical machine which is able to accommodate with sufficient resource. If the constraint is still violated, the algorithm begins a procedure that switches virtual machines to physical machines until either the constraints are all satisfied or no more moves can be made to produce a possible feasible solution, which usually means keeping the original placements.
The second stage of the heuristic serves the primary purpose of consolidating VMs in order to sleep more PMs, reducing energy consumption. The resource utilization of PMs are summed up to draw a comparison between origin to destination. PMs are chosen in ascending order as prospective candidates if the VMs residing on them could be potentially migrated to the remaining PMs. Once the candidate PMs are selected, the heuristic chooses one out of the rest of active PMs to host incoming VMs.

Then the VMs currently residing on a given candidate PM migrates tentatively to check whether the following conditions are satisfied: no resource utilization constraints are violated and the after-migration energy consumption is less than the initial placement energy consumption. As long as one of the conditions fails, the given solution will not be sufficient to improve with the tentative candidate consideration. Along with the consolidation process, when encountering a resource constraint violation, the algorithm will start from the resource exceeding the most ($H_{CPU}$, $H_{MEM}$, $H_{HD}$ or $H_{BW}$), and attempt to reach a solution without exceeding the limits down the road. All the energy terms in Eq. (3.9) are considered in this phase of the algorithm.

### 3.4 Testbed Implementation

We have built a real virtualized data center testbed to evaluate our design and ensure it can be practically applied to real world data centers. Currently we have four physical servers to host virtual machines, each configured with i7 3770 CPU, 16GB DDR3 memory. Each physical server is virtualized using VMware ESXi5 hypervisor. We have deployed 20 Ubuntu 12.04 LTS Linux virtual machines in this data center. Each VM is equipped with an iSCSI network storage and can be accessed by every physical server. A third server is used to host the heart of the data center vCenter, which manages all the VMs and hypervisors. vCenter is also responsible for sending out migration command once VM placement decisions are made. Two additional virtual servers are used to provide DNS, Active Directory Domain and network
Algorithm 3 Energy-saving VM Placement

**Input:** Period, \( L' \), \( N_{VM} \), \( N_{PM} \), \( P \), \( P_{active} \), \( P_{sleep} \), \( U_{CPU} \), \( U_{MEM} \), \( U_{HD} \), \( U_{BW} \), \( T_{migrate} \), \( H_{CPU} \), \( H_{MEM} \), \( H_{HD} \), \( H_{BW} \) and \( P_{migrate} \).

**Output:** \( L', G \) and \( O \).

**STAGE 1:** Feasible Solution Initialization

1. while There exists a resource constraint violation do
2. Perform virtual machine migration to find a feasible solution;
3. if A feasible solution cannot be found then
4. Adopt the alternative for operation;
5. break;
6. end if
7. end while

**STAGE 2:** Virtual Machine Consolidation

8. repeat
9. Seek a better solution to consume energy at a lower level;
10. until The solution cannot be improved.
11. return \( L', G \) and \( O \).

storage services. The following provides detailed information about the hardware and software setup of the testbed:

- vCenter Server 5.0: Intel Core i7-3770@3.40GHz, 4 GB RAM, runs 64-bit Windows 2008 Server R2.

- vCenter Database: Intel Core i7-3770@3.40GHz, 4 GB RAM, runs 64-bit Windows 2008 Server R2 and Microsoft SQL Server 2005.

- ESX 5.0 Servers: Intel Core i7-3770@3.40GHz, 32 GB RAM.

- Network: 1Gbps vMotion network configured on a private LAN.

- Storage: 2 1TB iSCSI storage hosted by 2 Windows 2008 Server R2, shared by all the hosts.

We configure a Hadoop cluster built with Apache Hadoop 2.2.0 on top of our testbed data center to perform execution of MapReduce benchmarks. The cluster is composed of one master
node and 20 computing nodes running Ubuntu 12.04. Each node has 4 GB of memory and 40 GB hard disk. The Hadoop configuration uses the default settings and runs with Oracle JDK 1.7. The estimation of resource usage of a specific VM is based on the VM’s history resource usage as all applications have program phases [63] that last for a period of time. With the characterization of the workloads and benchmark suite used in this paper, the workload fluctuations range from seconds to minutes, which are actively monitored by the CoolCloud data center. In this design, we use one minute as the threshold for a stable program phase and the threshold for initiating VM migration/remapping.

### 3.5 Experiment Result

We first evaluate the energy conserving capability of our optimization framework (the ILP design) to demonstrate that optimal dynamic VM placement can be achieved. Secondly, we demonstrate that the heuristic design is capable of achieving near optimal results. Thirdly, we use simulation to demonstrate that the heuristic design can scale well to large scale server clusters.

In order to thoroughly examine whether the dynamic VM placement decisions could effectively result in a balanced and energy aware data center, long-running and fluctuating workloads are required to trigger the VM migration. These workloads include: Apache ab, Phoronix Test Suite [69] and HiBench [85]. HiBench is a widely-used benchmark suite for Hadoop provided by Intel to characterize the performance of MapReduce based data analysis running in data centers. While the benchmark programs are running, our dynamic VM placement software will keep monitoring the VMs and servers to make migration decisions when necessary. At the same time, we keep record of each physical server’s resource utilization and power consumption for a one hour period. We run all experiments three times and use the average as the result.
For the evaluation of energy saving capabilities of VMware DRS [21] and our heuristic algorithm, the same testbed and workloads are used. We compare these three designs in regard of their abilities to balance workloads, server resources and their energy saving abilities. The results of network utilization, imbalance score and power consumption of each design are compared to demonstrate their overall performance.

The power consumption of each physical server is measured based on the work in [17, 62], where the full-system average power is approximately linear with respect to CPU utilization as given in eq. (3.10). It has proven to be an accurate way of measuring server power consumption especially in a data center environment where the total power consumption is an aggregation over a large number of servers.

\[
P_{Total} = P_{Dynamic} \cdot U_{Avg} + P_{Idle}
\]  

(3.10)

In eq (3.10), \( P_{Total} \) is the total power consumption of the server, \( P_{Dynamic} \) is the dynamic power consumption of the CPU, \( U_{Avg} \) is the average CPU utilization and \( P_{Idle} \) is the power consumption when CPU is idle. In our experiment, all the metrics on the right side of eq.(3.10) are measured using the Intel Power Gadget [1].
3.5.1 Evaluation on Testbed

In the following experiment result charts, note that CoolCloud is the proposed optimization design, where CoolCloud(I) represents the ILP design and CoolCloud(H) represents the heuristic design.

Figure 3.2 shows the network utilization of each server while the testbed data center is managed by VMware DRS. As we can see there are big differences of network bandwidth consumption of each server. For example, within the 15 minutes examining period, server 1 only consumes less than 100 Kbps of bandwidth. Server 4 on the other hand, consumes more than 700 Kbps of bandwidth. This is because DRS does not balance the network resource utilizations across servers. This is especially harmful when several VMs that all require high network bandwidth are placed on the same server. This design flaw causes resource wastage: due to the bottle neck of one resource, other resources can not be fully utilized. For example, in the case of unbalanced network utilization, if a PM runs out of network bandwidth, even if it still has large amount of remaining CPU or memory resource, it is unlikely to accommodate any more VMs.
On the other hand, Figure 3.3 shows the network utilization of each servers while the testbed data center is being managed by our dynamic VM placement design. To demonstrate the effectiveness of our design and to save space at the same time, we only show the result for CoolCloud ILP, since the result for Heuristic is similar. The network utilization starts unbalanced with server 4 having heavy network traffic (1200 Kbps) and server 2 having very little network traffic (0 Kbps). Our optimization model quickly detects this imbalance and provides the optimal placement solution. In about 3 minutes, the migrations are complete and the network bandwidth consumptions are balanced across all servers. This demonstrates our design solves the unbalanced issue in DRS, eliminating potential network bandwidth bottlenecks.

Figure 3.4 shows the power consumption of the data center managed by No Migration, DRS, CoolCloud(I) and CoolCloud(H). Each case is monitored in a 60 minutes time period. The data center starts with the same workload and initial VM placement. The result shows both DRS and our design are capable of achieving power savings. DRS can provide 15.5% power savings on average compared to the settings where no management scheme is used at all. CoolCloud(I) and CoolCloud(H) achieved 28.6% and 28.3% power savings respectively when comparing with the case of no management scheme used, and this is over 15% gain of
Figure 3.5: CoolCloud Performance Evaluation

power savings compared with \textit{DRS}. The power consumption measured here is the result of taking all costs including the cost of live migration into consideration.

Both \textit{DRS} and our design provide power savings by turning off under utilized servers, however our design is capable of achieving the maximum power savings. This is because \textit{DRS} mainly focuses on balancing CPU resource, and it only periodically (every 5 minutes) checks if any server is under utilized. This periodic checking may miss some energy saving opportunities due to the fluctuation of workloads. Further more, DRS does not provide the balancing of memory or network bandwidth across servers. This implies that some servers cannot be turned off due to resource wastage which leads to waste of energy. On the other hand, our design has an objective of minimizing energy consumption and all aspects of server resources are being considered. This creates a well balanced data center in regard of all resources, thus more servers can be turned off to achieve more energy savings. Notice that our design constantly monitors the server resource utilization in a pro-active fashion, thus responding quickly to the workload fluctuations and seizing every energy saving opportunities.

Figure 3.5 provides the performance evaluation of CoolCloud. In this experiment, we measure the execution time for four benchmark programs from HiBench, i.e., WordCount,
Sort, PageRank and Kmeans. For WordCount, the execution time is about the same across all three configurations, i.e., 132s for No Migration, 138s for CoolCloud and 130s for DRS. CoolCloud requires slightly longer time to complete execution due to the overhead of live migration. However for PageRank and Kmeans (825s for No Migration, 684s for CoolCloud and 756 for Kmeans), CoolCloud demonstrates significant lower execution time compared to No Migration and DRS. This is because No Migration can not resolve the resource contention issue experienced by VMs, and DRS only reacts to this issue every 5 minutes. On the other hand, CoolCloud is able to detect the resource contention proactively and respond quickly by initiating VM migration to resolve this issue. Note that the cost for live migration is fully considered in both the optimization model and the heuristic. The time duration for live migration typically ranges from several seconds to tens of seconds depending on the memory footprint of the VM. The small performance degradation comes from the live migration overhead and affects the applications performance running on that specific VM. CoolCloud prioritizes VMs’ with smaller memory footprints for migration thus significantly reduces the overall migration overhead.

3.5.2 Evaluation through Simulation

The evaluation result of the ILP and heuristic designs against the test bed data center has proven to provide better energy conservations compared to VMWare’s Dynamic Resource Planning design. In this section, we demonstrate that the heuristic design can be effectively applied to large-scale clusters to provide energy savings. To thoroughly evaluate the heuristic design, we designed an hybrid approach that combines profiling VM data from the test bed and simulating a large-scale cluster using the collected data.

3.5.2.1 VM Profiling

The goal of VM profiling is to generate large numbers of VMs with runtime information and feed these VMs’ runtime info as inputs to the ILP and heuristic to evaluate their perfor-
Figure 3.6: Energy consumption for ILP and Heuristic

Figure 3.6: Energy consumption for ILP and Heuristic

Figure 3.6: Energy consumption for ILP and Heuristic

3.5.2.2 Performance Comparison of CPLEX and Heuristic Algorithm

The simulation for the ILP design and the heuristic design are carried out with the same system configuration: A server with 2 Intel Xeon x5650 CPUs which has 24 virtual cores, Red Hat Enterprise Linux Workstation 6.6 (Santiago) with 2.6 kernel, and the total amount of memory is 47 GB. The optimization ILP formulation is solved by IBM CPLEX 12.5.

Figure 3.6 displays the energy consumption result for the ILP design and the Heuristic design when the number of virtual machines in the data center ranges from 20 to 1000. In
the case of 1000 VMs, with the management of ILP, the data center energy consumption is 5280kJ and this number is 5401kJ for applying the heuristic design. This means the solution provided by the heuristic design only differs 2.3% from the optimal result. Overall, this result demonstrates that the heuristic design can provide solutions with only slight degradation on energy savings compared to the optimal ILP design.

Figure 3.7 displays the computation time of ILP and Heuristic. In the case of 20 VMs, ILP and Heuristic take comparable time for calculation with 180ms and 375ms respectively. At the point of 50 VMs, the computation time is about the same with 630ms and 661 respectively. However when there are more than 50 VMs, the computation time for solving ILP grows dramatically as the number of VMs increases. In the case of 1000 VMs, the computation time for ILP and Heuristic are 680s and 22s respectively. This result demonstrates that the heuristic design is highly computational efficient when it comes to large-scale clusters.

Overall, the simulation result shows that the heuristic design can provide near optimal solutions for energy savings and it is highly computational efficient making it a practical solution for large-scale data centers.
3.6 Related Work

Earlier work mostly focuses on improving resource utilization and load balancing of VMs across physical servers [85]. Timothy et al. [85] propose a VM mapping algorithm called Sandpiper to detect hotspots and relocate VMs from overloaded servers to under-utilized ones. When a migration between two servers is not directly feasible, Sandpiper can identify a set of VMs to interchange in order to free up sufficient amount of resources on the destination server. This approach is able to solve specific replacement issues but requires extra memory space for interim hosting of VMs. This process also needs extra rounds of migration and may affect system performance.

Static VM placement methods [88, 53, 78] are effective for initial VM placements. However, these approaches do not consider future dynamic workload changes that may need VM remappings. Jing et al. [88] propose a multi-objective virtual machine placement algorithm that simultaneously minimize power consumption, resource wastage and thermal dissipation. Xin et al. [53] also considers physical resources as multi-dimensional and propose a multi-dimensional space partition model to determine the mapping of VMs and PMs.

Furthermore, Bobroff et al. [8] uses first-fit approximation to solve the VM placement problem focusing on CPU utilization. Fabien et al. [29] formulate VM placement into a constraint satisfaction problem to minimize the number of physical machines. This work considers uni-processor computers and assumes each PM can only host one active VM. Hien et al. [65] extends [29] and considers CPU and RAM as constraints.

3.7 Conclusion

This paper presents a dynamic virtual machine placement framework to manage the mappings of VMs to physical servers. This framework tackles the problem of finding the most energy efficient way (least resource wastage and least power consumption) of placing the VMs considering their fluctuating workloads and resource requirements. With all resource con-
straints and migration cost taken into account, the design is implemented and evaluated against a real testbed. It is proven that CoolCloud is highly scalable and can be practically applied to enterprise data centers for great energy efficiency.
CHAPTER 4. TAMING ENERGY CONSUMPTION IN DATA CENTERS WITH DYNAMIC VIRTUAL MACHINE PLACEMENT

This paper is submitted to *IEEE Transactions on Computers*

Zhiming Zhang\(^1\), Chan-Ching Hsu and J. Morris Chang\(^2\)

abstract

With the continuing growth of cloud computing services, energy consumption has become one of the most challenging issues in data center environments. With the support of today’s virtualization technology, the efficiency and flexibility of data center management can be greatly enhanced, creating many energy saving opportunities. However, effective energy aware design is a non-trivial task, considering the size of the data center, the dynamic fluctuation of workloads and the variation of computing resource requests. In this paper, we propose CoolCloud: a practical solution for managing the mappings of VMs to physical servers. This framework solves the problem of finding the most energy efficient way (least resource wastage and least power consumption) of placing the VMs considering their resource requirements. A real testbed data center implemented with industry product VMware vSphere 5 is used to evaluate the proposed framework. We run various cloud computing workloads including web service applications, big data benchmarks and software containers on VMs to demonstrate that CoolCloud can effectively improve data center performance and energy efficiency under dif-

\(^{1}\)Primary researcher and author.

\(^{2}\)Author for correspondence.
ferent cloud computing scenarios. A simulation study shows the proposed design scales well to large size data centers.

4.1 Introduction

Rapid growth of cloud computing services have led to creation of large scale enterprise data centers. In 2013, U.S. data centers consumed an estimated 91 billion kilowatt-hours of electricity [66], which is 2% of the total U.S. electricity consumption. This number is projected to increase to 140 billion kWh by 2020. How to reduce energy cost in a data center has now become one of the most challenging issues for cloud computing service providers.

Virtualization technology has been adopted to enhance efficiency and flexibility in data center management and resource provisioning. With the support of virtualization, resources of a single server can be divided into multiple isolated execution environments. This allows one physical machine to host several virtual machines (VMs), which helps to achieve higher per-server utilization that results in higher energy efficiency.

A virtualized data center environment provides a shared hosting infrastructure to customers who need server resources to run their applications. All the applications run inside virtual machines (VMs) which are provisioned and managed on-demand. VMs’ resource utilization (CPU, memory, network, etc.) must be constantly monitored and the data center manager must respond to the changing on-demand resource requests from applications and determine which physical server a VM should be placed on. This is a time consuming task that can not be performed by human operators in a timely fashion considering the complexity and size of the data center.

In recent years, virtualized data center management has brought a great amount of research interest. Earlier work mostly focuses on improving resource utilization and load balancing of VMs across physical servers [74, 90, 32, 18]. Recently, more work [89, 80, 25, 21] is being proposed to improve energy efficiency. However, most of the extant approaches have either
one or several of the following limitations: 1. only consider solving one specific problem or focus on one aspect of optimization, e.g., balancing VMs across servers, eliminating hot spots, minimizing power consumptions, maximizing resource utilizations, etc; 2. the optimization models do not scale to the size of large enterprise data centers; 3. only focus on one or two server resources at the same time, e.g., CPU, memory or network bandwidth. Such solutions may perform well on the server resource(s) being considered and leave potential performance bottlenecks on the resource(s) being left out; 4. only provide the initial placement of VMs without taking care of the runtime workload fluctuations; 5. evaluations are mostly carried out through simulation studies that oversimplifies the real scenarios in data center management; 6. limited workload or benchmark selection that do not well represent today’s cloud computing environment. These limitations hinder recent works from being practically applied to enterprise data centers for higher energy efficiency.

In this paper, we address the above discussed challenges and propose CoolCloud: a dynamic virtual machine placement framework which finds the most energy efficient way of placing VMs according to their resource requirement. We name our design CoolCloud given its ability to cool down the data center that provides a more energy efficient cloud. CoolCloud actively monitors each VM and collects their runtime resource utilization which includes: CPU, memory, network and storage. An integer linear programming (ILP) optimization model is proposed to take the VM resource utilizations as input and provide the optimal placement solution as output. The objective of this ILP model is to minimize data center energy consumption without affecting each VM’s performance.

A heuristic with low computation complexity based on quick sort and greedy algorithm is designed to solve the placing problem timely. The efficient heuristic is key to making CoolCloud scalable to large size data centers. Maximum energy saving is achieved through consolidating VMs to the least number of servers and turning remaining servers to sleep mode. Note that the resource monitoring process collects server resource utilization of all aspects, which is key to eliminating resource wastage or performance bottlenecks. CoolCloud not
only provides the initial placement of VMs, but proactively monitors runtime workload fluctuations and provides new placement solutions in case of service level agreement violations or a more energy efficient placement is discovered. A new placement map will be generated when a new solution is discovered, VMs will be migrated to their designated server location according to the placement map. Live migration is used in CoolCloud to move VMs around which minimizes downtime and interruption to users. The performance and energy cost of live migration are both considered in our optimization model and heuristic design.

CoolCloud is implemented in Java with VMware vSphere 5 SDK. To thoroughly evaluate our design, we build a testbed data center with vSphere 5 suite including ESXi 5 and the vCenter management platform. We choose workloads from web service applications to test the effectiveness of CoolCloud in regards of energy savings and performance. We also implement Hadoop in our testbed to evaluate CoolCloud in today’s popular Hadoop environment. As usage of software containers is becoming a popular way to develop and deploy cloud applications, we have built many Docker software containers as workloads to run in our data center. We conduct an experiment to demonstrate how CoolCloud performs when managing VMs hosting Docker containers. The scalability of the heuristic design is studied with a simulation which provides a comparison between the heuristic and the ILP model in terms of the amount of time used to find the placing solutions. We compare our design with VMware’s Distributed Resource Scheduler (DRS), experiment result demonstrates CoolCloud achieves better overall performance and energy savings. In summary, the main contributions of our work are:

- We propose CoolCloud with the objective to minimize energy consumption. CoolCloud actively monitors workload runtime fluctuations, and provides dynamic placement solutions.

- CoolCloud considers server resource utilization in all aspects that eliminates energy wastage and performance bottlenecks caused by resource wastage.
• We provide a heuristic design based on simple quick sort and greedy algorithm that achieves near-optimal energy savings with low computation complexity. This makes CoolCloud a practical solution for large size data centers.

• We build a real testbed data center to evaluate CoolCloud. We choose workloads from web service applications, big data benchmarks, i.e., HiBench to Docker software containers that represent today’s cloud computing environment to thoroughly evaluate CoolCloud.

• We conduct a comparison with industry leading design VMware’s DRS to demonstrate the effectiveness of CoolCloud in regard of performance and energy savings.

The remaining of the paper is organized in the following sequence. Related work is given in Section 4.2. We provide our VM placement optimization model in Section 4.3. Section 4.4 introduces the computation efficient heuristic design. The implementation of our framework is provided in section 4.5. Section 4.6 demonstrates the experiment results and Section 4.7 concludes this paper.

4.2 Related Work

Virtualized data center management has gathered a great amount of research interests in the past few years. Recent studies focus on improving server resource utilizations, meeting power budgets, balancing workloads among servers and reducing any energy related costs. We have done an extensive study of past works to inspire our design. A brief discussion of past achievements and limitations is given as follows.

Timothy et al. [85] propose a VM mapping algorithm called Sandpiper to detect hotspots and relocate VMs from overloaded servers to under-utilized ones. When a migration between two servers is not directly feasible, Sandpiper can identify a set of VMs to interchange in order to free up sufficient amount of resources on the destination server. This approach is able to
solve simple replacement issues but requires extra memory space for interim hosting of VMs. This process also needs extra rounds of migration and may affect system performance. Singh et al. [75] propose a load balancing algorithm called VectorDot for handling the hierarchical and multi-dimensional resource constraints in virtualized data centers. The proposed algorithm is evaluated on a real data center testbed built with VMware ESX servers and network attached storages.

Grit et al. [24] consider some VMs replacement issues for resource management policies in the context of a system for on-demand leasing of shared networked resources in server clusters. When a migration is not directly feasible, due to sequence issues, the VM is suspended using suspend-to-disk. Once the destination server becomes available, the VM resumes. In our work, when migration is not feasible, we first try to find the delay-tolerant VM and suspend it to release server resource for other time-sensitive VMs. When server resource becomes available, the suspended VM resumes. Electricity price is also considered for determining when to resume the suspended VM in order to minimize cost.

Jing et al. [88] propose a multi-objective virtual machine placement algorithm that simultaneously minimize power consumption, resource wastage and thermal dissipation. Xin et al. [53] also consider physical resources as multi-dimensional and propose a multi-dimensional space partition model to determine the mapping of VMs and PMs. [77, 64, 9, 88, 53] are static VM placement methods that only consider the initial VM placement and do not consider future dynamic workload changes that may need VM remappings. Shrivastava et al. [73] consider the inherent dependencies between VMs comprising a multi-tier application. They propose a scheme called AppAware to determine VM placement that can greatly reduce network traffic considering the interaction between applications running on different VMs.

Meng et al. [63] consider the network traffic and bandwidth as factors that may affect system performance. They optimize VM placement based on traffic patterns and communication distances. VMs with mutual bandwidth usage are assigned to PMs. Cost-aware workload placement is also gathering wide interest in data center operations. [6, 61, 82, 51, 57] propose to
reduce data center operational cost by exploiting electricity price differences across regions. Workload can be migrated to a data center where resource is sufficient and energy price is low. However, migration cost and service level agreement are not considered. Furthermore, short running workloads do not make sense in this scenario. [7] focus on how to use green energy to power data centers.

The works we studied above all take advantage of server virtualization and provides improved and balanced resource utilization. However, these works do not consider the cost of VM migration while making VM remapping decisions which may lead to undesirable results. In our work, we take this actual cost of live migration into account, and we put this cost in terms of time and energy consumption into our model to decide whether migration is indeed beneficial. Another weakness is that some works use simulation to evaluate their proposed algorithm or schemes which oversimplifies the actual dynamic changes of workloads in a real data center. In our work, we build a real virtualized data center testbed comprised of VMware ESXi servers, ISCSI network attached storages which provides a comprehensive evaluation of our design.

Bobroff et al. [8] use first-fit approximation to solve the VM placement problem focusing on CPU utilization. Fabien et al. [29] formulate VM placement into a constraint satisfaction problem to minimize the number of physical machines. This work only considers uni-processor computers and assumes each PM can only host one active VM. The VM placement problem is oversimplified. In our work, the VM placement problem is not constrained by uni-processor computers. Instead, the constraints come from whether available resource on PM can satisfy VM SLA or resource requirement. Hien et al. [65] extends [29] and considers CPU and RAM as constraints. However their implementation assumes these two inputs are known in advance. In our work, we use VM history to predict its future resource needs. We also plan to study the autocorrelation and periodic attributes of VM history for more accurate prediction.
4.3 System Model

The proposed design shown in Figure 4.1 includes three major components. The first component is responsible of collecting runtime resource utilizations of each VM. These resources include CPU, memory, network and storage. The second component is a linear programming optimization model that provides the optimal VM placement solution. The objective function of this model is to minimize data center energy consumption without affecting each VM’s performance. The model takes each VM’s resource requirements as its constraints to guarantee performance. The third component is a commander responsible of sending out VM migration commands based on the placement solution from the optimization model. Live migration [46] is used here so service of each VM will not be interrupted during dynamic placements. Our design also considers the VM migration cost which is often ignored in past works.
4.3.1 Live Migration

One major benefit of virtualization is that it enables server consolidation and provides better multiplexing of data center resources across VMs. Carefully placing VMs on physical servers (PSs) based on their resource needs can greatly reduce data center management cost and energy consumption. VM live migration is an emerging technology that enables virtual machines to migrate from one physical server to another with little service downtime. As shown in Fig. 4.2., VM5 is migrated from PM2 to PM3. Live migration allows us to dynamically remap VMs to PSs based on the changing workload. For example, when a PS is overloaded, VMs can be moved out and migrate to a less loaded PS to guarantee performance. On the other hand, when there are several under-utilized PSs, VMs can be consolidated to a fewer number of PSs thus providing energy savings.

However, live migration does not come free. Live migration works by transferring VM’s architecture states and memory data from its original host to its destination host. This process consumes extra CPU and network resource and may harm performance thus must be taken
### Table 4.1: Definitions of Important Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Number of physical machines to serve virtual machines</td>
</tr>
<tr>
<td>$M$</td>
<td>Number of virtual machines</td>
</tr>
<tr>
<td>$P_{\text{active}}$</td>
<td>Basic power level of physical machines in active mode</td>
</tr>
<tr>
<td>$P_{\text{sleep}}$</td>
<td>Power level of physical machines in sleep mode</td>
</tr>
<tr>
<td>Period</td>
<td>Time period for which the solution pertains</td>
</tr>
<tr>
<td>$P_{\text{migrate}}$</td>
<td>Power level for VM $m$ migrating to PM $n$</td>
</tr>
<tr>
<td>$T_{\text{migrate}}$</td>
<td>Time for VM $m$ migrating to PM $n$</td>
</tr>
<tr>
<td>$H_{\text{CPU}}$</td>
<td>Limit on CPU utilization of physical machines</td>
</tr>
<tr>
<td>$H_{\text{MEM}}$</td>
<td>Limit on memory utilization of physical machines</td>
</tr>
<tr>
<td>$H_{\text{HD}}$</td>
<td>Limit on hard disk utilization of physical machines</td>
</tr>
<tr>
<td>$H_{\text{BW}}$</td>
<td>Limit on network bandwidth utilization of physical machines</td>
</tr>
<tr>
<td>$N_{\text{VM}}$</td>
<td>Set of VMs, $</td>
</tr>
<tr>
<td>$N_{\text{PM}}$</td>
<td>Set of PMs, $</td>
</tr>
<tr>
<td>$U_{\text{CPU}}$</td>
<td>Virtual machine CPU utilization, $U_{\text{CPU}} = {U_{m\text{CPU}}, \forall m \in N_{\text{VM}}}$</td>
</tr>
<tr>
<td>$U_{\text{MEM}}$</td>
<td>Virtual machine memory utilization, $U_{\text{MEM}} = {U_{m\text{MEM}}, \forall m \in N_{\text{VM}}}$</td>
</tr>
<tr>
<td>$U_{\text{HD}}$</td>
<td>Virtual machine hard disk utilization, $U_{\text{HD}} = {U_{m\text{HD}}, \forall m \in N_{\text{VM}}}$</td>
</tr>
<tr>
<td>$U_{\text{BW}}$</td>
<td>Virtual machine network bandwidth utilization, $U_{\text{BW}} = {U_{m\text{BW}}, \forall m \in N_{\text{VM}}}$</td>
</tr>
<tr>
<td>$E$</td>
<td>Total system energy consumed</td>
</tr>
<tr>
<td>$L$</td>
<td>Placement matrix (decision variable), $L = (l_{mn})_{M \times N}$</td>
</tr>
<tr>
<td>$G$</td>
<td>Migration matrix (decision variable), $G = (g_{mn})_{M \times N}$</td>
</tr>
<tr>
<td>$O$</td>
<td>Operation mode vector (decision variable), $O = (o_n)_{1 \times N}$</td>
</tr>
</tbody>
</table>

into consideration before migration is applied. Virtual machine placement algorithms have been proposed to enhance the performance and energy efficiency of a data center.

#### 4.3.2 VM Placement Problem Formulation

The following sections presents the optimization model for the virtual machine replacement problem. We first provide the important definitions of symbols used in our formulation given in Table 4.1. Let $N_{\text{VM}} = \{VM_0, \ldots, VM_{M-1}\}$ be the set of virtual machines with cardinality $|N_{\text{VM}}| = M$. Similarly, let $N_{\text{PM}} = \{PM_0, \ldots, PM_{N-1}\}$ be the set of physical machines with cardinality $|N_{\text{PM}}| = N$. The problem is defined as follows:

Given (1) the power and time requirements of virtual machine $m$ to run on physical machines $n$, $P_{mn}$ and $T_{mn}$, (2) the migration cost of each virtual machine in power, $P_{\text{migrate}}$, and
in time, $T_{mn}^{\text{migrate}}$, (3) the physical machine capacity in terms of cpu, memory, hardware and bandwidth utilization, $H_n^{\text{CPU}}$, $H_n^{\text{MEM}}$, $H_n^{\text{HD}}$ and $H_n^{\text{BW}}$, (4) the virtual machine utilization requirements for CPU, $U_m^{\text{CPU}}$, memory, $U_m^{\text{MEM}}$, hard disk, $U_m^{\text{HD}}$ and bandwidth, $U_m^{\text{BW}}$, (5) the active and sleep mode power use of the physical machines, $P_n^{\text{active}}$ and $P_n^{\text{sleep}}$, the problem is to minimize the system energy consumption, denoted as $E$, by placing virtual machines on physical machines PMs, deploying VMs in active mode with the consideration of the migration cost. The output includes the virtual machine destinations, migration indicators and operation mode of physical machines specified as, $l_{mn}$, $g_{mn}$ and $o_n$.

4.3.2.1 Decision Variables

The decision variables of the placement problem are presented by two matrices $L$, $G$ and a vector $O$. $L = (l_{mn})_{M \times N}$ is the virtual-physical machine incidence matrix, or the placement matrix and $G = (g_{mn})_{M \times N}$ is the virtual machine migrating incidence matrix, or the migration matrix; $O = (o_n)_{1 \times N}$ is the physical machine activation incidence vector, or the operation mode vector. The value of a decision variable is determined as follows.

$$l_{mn} = \begin{cases} 
1, & \text{VM } m \text{ is placed on PM } n, \\
& \forall m \in \mathbb{N}_V M, n \in \mathbb{N}_P M; \\
0, & \text{otherwise},
\end{cases}$$

$$g_{mn} = \begin{cases} 
1, & \text{VM } m \text{ is migrated to PM } n, \\
& \forall m \in \mathbb{N}_V M, n \in \mathbb{N}_P M; \\
0, & \text{otherwise},
\end{cases}$$

$$o_n = \begin{cases} 
1, & \text{PM } n \text{ is in active mode}, \forall n \in \mathbb{N}_P M; \\
0, & \text{otherwise},
\end{cases}$$

When $l_{mn}$ is asserted, the physical machine $n$ serves the virtual machine $m$, provisioning with the power, $P_{mn}$; $o_m$ has to set a value of one correspondingly. If VM $m$ does migrate to PM $n$, the migration cost involves power, $P_{mn}^{\text{migrate}}$ and time, $T_{mn}^{\text{migrate}}$ for the procedure. It
conceivable that a physical machine operating actively \( (o_m = 1) \) consumes energy at a much higher level than in sleep mode.

### 4.3.2.2 Placement Constraint

Each virtual machine can be served by only one physical machine, and it must be placed on one of the physical machines to have the resource granted toward it. The following constraint essentially set up this condition to satisfy.

\[
\sum_{n \in \mathbb{N}_{PM}} l_{mn} = 1, \forall m \in \mathbb{N}_{VM}, \quad (4.1)
\]

For an individual virtual machine \( m \), if the result of \( \sum_{m \in \mathbb{N}_{VM}} l_{mn}, \forall n \in \mathbb{N}_{PM} \) is equal to 0, it means the physical machine \( n \) operates in sleep mode and no virtual machines will run on it.

The idea of virtual machine migration is put into action if a virtual machine is placed on a physical machine \( n \) different from the one by which it is currently served before the optimal policy is made. Namely, for a machine \( m \), its next placement, \( l_{mn} = 1 \) and its current placement, \( l'_{mn} = 0 \), which is given information initially, are compared to represent that the virtual machine \( m \) is migrated to the physical machine \( n \) from other physical machine. Constraint (4.2) gives the value of \( g_{mn} \) by the comparison of the two states, \( l_{mn} \) and \( l'_{mn} \).

\[
l_{mn} - l_{mn} \cdot l'_{mn} = g_{mn}, \forall m \in \mathbb{N}_{VM}, n \in \mathbb{N}_{PM}, \quad (4.2)
\]

### 4.3.2.3 Resource Constraints

There are constraints which define resource utilization in the physical machine domain. Virtual machines on each physical machine in the solution need guaranteed service levels. The service level are categorized by four aspects: CPU, memory, hard disk and bandwidth utilization, which haven’t considered in the past work and firstly are jointly discussed here. Essentially, the deployed resources of a physical machine cannot exceed a specified utilization level. Based on service level requirements, the resource utilization of physical machines varies.
The resource constraints are provided in the following, where $U_m$ indicates the utilization of virtual machine $m$:

\[
\sum_{m \in \mathbb{N}_{VM}} (l_{mn} \cdot U_{m}^{CPU}) \leq H_{n}^{CPU}, \forall n \in \mathbb{N}_{PM}, \tag{4.3}
\]

\[
\sum_{m \in \mathbb{N}_{VM}} (l_{mn} \cdot U_{m}^{MEM}) \leq H_{n}^{MEM}, \forall n \in \mathbb{N}_{PM}, \tag{4.4}
\]

\[
\sum_{m \in \mathbb{N}_{VM}} (l_{mn} \cdot U_{m}^{HD}) \leq H_{n}^{HD}, \forall n \in \mathbb{N}_{PM}, \tag{4.5}
\]

\[
\sum_{m \in \mathbb{N}_{VM}} (l_{mn} \cdot U_{m}^{BW}) \leq H_{n}^{BW}, \forall n \in \mathbb{N}_{PM}, \tag{4.6}
\]

For physical machines, the resource utilization level is limited to be not over a specific value, for example, $H_{n}^{CPU}$ is the maximum CPU utilization level, which could be a measurement of percentage. A set-up percentage less than 100% leaves the margin for new arrival tasks. Energy can be consumed at diverse power levels and the execution time are different from virtual machines.

### 4.3.2.4 Operation Mode Constraints

The following two constraints define in which mode a physical machine will run. If it is a physical machine operating in active mode, $o_m$ is equal to one for PM $m$.

\[
o_n \leq \sum_{m \in \mathbb{N}_{VM}} l_{mn}, \forall n \in \mathbb{N}_{PM}, \tag{4.7}
\]

\[
l_{mn} \leq o_n, \forall m \in \mathbb{N}_{VM}, n \in \mathbb{N}_{PM}, \tag{4.8}
\]

Constraint (4.7) and (4.8) together satisfy that a physical machine operate in active mode as long as in charge of supplying service to any virtual machine, or the physical machine is able to run inactively with no virtual machines to serve.

### 4.3.2.5 Objective Function

We define the energy consumption as a summation of the virtual machine execution energy, migration energy, active physical machine energy and sleep physical machine energy, which
are termed below respectively. The next expression presents the energy consumed as a whole, where Period is the predefined time period.

$$\begin{align*}
\min E &= \sum_{m \in VM, n \in PM} (P_{mn} \cdot \text{Period} \cdot l_{mn}) \\
&\quad + \sum_{m \in VM, n \in PM} (P_{migrate} \cdot T_{mn} \cdot g_{mn}) \\
&\quad + \sum_{n \in PM} (P_{active} \cdot \text{Period} \cdot o_n) \\
&\quad + \sum_{n \in PM} [P_{sleep} \cdot \text{Period} \cdot (1 - o_n)].
\end{align*}$$

(4.9)

Eq. (4.9) is the objective function optimizing the total energy consumption of the data center. $P_{mn}$ denotes the required power level of the virtual machine $m$ to operate on the physical machine $n$. The required power levels and time for migration are related to the resource usage, different from virtual machine to virtual machine. The idea is to find out how to place virtual machines onto physical servers in the way that energy cost is minimized. The optimization model takes advantage of the migration feature and consolidates virtual machines onto less number of active physical servers. For each virtual machine, its resource needs are satisfied to guarantee performance by aforementioned constraints. The resource utilization on physical machines is tend to be utilized fully so less servers need to be active, saving energy.

### 4.4 Approach to Solve the Problem

Since the formulated problem in Section 4.3 is ILP, which is generally NP hard, obtaining the optimal solution can result in an exponential increase in computation time as the problem size grow, underlying the issue of scalability. Therefore, as a countermeasure against such adversity, we develop a heuristic algorithm which has the goal of offering a suboptimal solution and ease computational intensity. On the other hand, in the considered energy-aware data center environment, because the number of nodes (i.e., physical and virtual machines) changes and
resource requirements vary (i.e., CPU, memory, hard disk, and network bandwidth) frequently, a less expensive and less time-consuming approach is desired to optimize energy consumption in response to the dynamics; hence, a heuristic approach is favored to be implemented for solving the proposed optimization model in practice. The pseudocode is shown in Algorithm 4 and Algorithm 5, and is implemented using Java programming language. It takes the same input with what the optimization model takes into account; the algorithm determines the placement of virtual machines, virtual machine migrations, and operation mode of physical machines. In the following, we first discuss the algorithm design principles and then the details about the heuristic.

4.4.1 Design Principle

Two stages are devised in the algorithm with separate objectives. In the first stage, we want to determine whether a feasible solution exists prior to performing virtual machine consolidation, which is the second stage. Since the model specifies the maximum resource utilization of physical machines in each category (i.e., resource constraints), the algorithm checks if the current operation (i.e., $l'_{mn}$) violates the constraints. If a constraint is violated, the algorithm starts to find a feasible solution by performing virtual migration. The objective in this stage is to obtain a feasible solution as an initial solution to begin virtual machine consolidation. If no such feasible solution can be found, the algorithm determines no consolidation is necessary, and an alternative will be adopted when the problem is infeasible. In the following stage, the objective is looking for a better virtual machine placement decision in terms of energy consumption. Considering the difference between the operation energy in active and sleep mode of physical machines, switching physical machines into sleep mode results in reducing energy consumption. With such idea, placing virtual machines to a less number of physical machines is necessary to produce a better solution. Therefore, the attempt is to reach a new solution with an improved energy consumption by consolidating virtual machines to a smaller number of physical machines.
Algorithm 4 Energy-saving VM Placement

Input: \( \text{Period}, \mathcal{L}', N_{\text{VM}}, N_{\text{PM}}, P, P_{\text{active}}, P_{\text{sleep}}, U_{\text{CPU}}, U_{\text{MEM}}, U_{\text{HD}}, U_{\text{BW}}, T_{\text{migrate}}, H_{\text{CPU}}, H_{\text{MEM}}, H_{\text{HD}}, H_{\text{BW}} \) and \( P_{\text{migrate}} \).

Output: \( \mathcal{L}, \mathcal{G}, \) and \( O \).

1: Create resource utilization monitor;
2: \( \mathcal{L} = \mathcal{L}', \mathcal{G} = 0, \) and given \( O \);

STAGE 1: Feasible Solution Initialization
3: while The solution is not feasible do
4: for PMs with high utilization do
5: if The constraint is violated then
6: for VMs utilizing more resources do
7: Migrate VMs to PMs with lower utilization if constraints met;
8: end for
9: if The constraint still not satisfied then
10: for VMs utilizing more resources do
11: for PMs with lower utilization do
12: Exchange VMs if constraints satisfied;
13: end for
14: end for
15: end if
16: end if
17: end for
18: end while
19: if A feasible solution not found then
20: Adopt alternative for operation and leave;
21: end if

The algorithm is designed to be executed whenever the context changes, for example, virtual machines are done with its tasks, or more virtual machines are requested to perform more tasks, or in cases where input values have to be updated and the need of producing new solutions. However, the computation overhead might appear to be high, if the algorithm needs to perform constantly. To lower such overhead, \( \text{Period} \) can be used to control the time interval to run the heuristic, instead of producing solutions pro-actively. Overall, the computation complexity of the algorithm is \( O(MN(\log M)^2 \log N) \) in the worst case.
Algorithm 5 Energy-saving VM Placement (Continue Stage 1)

STAGE 2: Virtual Machine Consolidation
1: Create energy cost monitor;
2: repeat
3:     for Active PMs with lower total utilization do
4:         for Another active PMs with lower total utilization do
5:             Migrate all VMs on first PM to second PM;
6:             if Constraints violated then
7:                 for Violated constraints with higher utilization do
8:                     for VMs utilizing more resources do
9:                         for Another active PMs with lower total utilization do
10:                            Migrate VMs from second PM to third PM if not exceed the maximum;
11:                     end for
12:                 end for
13:             end if
14:         end for
15:     end for
16: end if
17: if Constraints violated then
18:     No solution updated;
19: end if
20: until The solution not improved.
21: return $\mathcal{L}$, $\mathcal{G}$ and $\mathcal{O}$.

4.4.2 Algorithm Details

Before going into the first-stage of the algorithm, we create resource utilization monitors in order to keep track of the resource utilization status of physical machines. As the algorithm goes along, these monitors are updated whenever a better solution is found. The utilization level is calculated based on the cumulative resource allocated to virtual machines on each physical machine, namely, $U_{CPU}$, $U_{MEM}$, $U_{HD}$ and $U_{BW}$. The current operation strategy is taken as the initial solution, i.e., $\mathcal{L} = \mathcal{L}'$, $\mathcal{G} = 0$, and $\mathcal{O}$ is given. The algorithm compares the monitored resource utilization levels with the maximum resource utilization values, i.e., $H_{CPU}$, $H_{MEM}$, $H_{HD}$ and $H_{BW}$. If any of the constraints is not satisfied, the solution is
not feasible, and a feasible solution needs to be found before entering the next stage of the algorithm.

In order to find a feasible solution, the heuristic tries migrating virtual machines virtually with the goal of reducing physical machine resource utilization. It first puts together and sorts utilization level in all categories of physical machines. Starting from the physical machine that consumes the highest utilized resource which exceeds the capacity (it can be CPU, memory, hard dish or bandwidth), on that physical machine we try two methods (line 4-7 and 8-11, respectively). First, we sort through each physical machine to find out which virtual machines utilize more respective resource, and search for destination physical machines with lower utilization level respectively. The algorithm then calculates the new utilization levels for physical machines involved in migration, e.g., source and destination physical machines. If after the above migration attempts, the constraint still cannot be satisfied, the heuristic will try the following migration strategy. Similarly, starting with virtual machines that utilize more respective resources, we search for physical machines with a lower resource utilization level. Then virtual machines on the selected physical machine will be picked for migration. The algorithm intends to migrate virtual machines with higher resource utilization so that the opportunity of finding a feasible solution with less migration cost is higher. As the resource monitor keeps track of utilization levels, as long as all constraints are satisfied, the program can move into the second stage. However if no feasible solution could be found after the procedure in the first stage, we determine there is no feasible solution to work on for energy saving procedure, and leave the routine.

The second stage of the heuristic serves the purpose of consolidating virtual machines in order to sleep physical machines, reducing energy consumption. Before performing the energy-saving procedure, we create the energy cost monitor to keep track of energy consumption of physical machines, migration cost and entire system status. In addition to respective resource utilizations, the total resource utilization of physical machines is summed up. Starting from physical machines having lower total utilization (first physical machine in line 17, sec-
ond in 18), the algorithm migrates virtual machines on the first physical machine to the second one, in order to switch off the first. However, if a constraint is violated after this migration, virtual machines will be migrated to other physical machines until constraints are satisfied, or until all other physical machines have been tried with no feasible solution reached. In the latter case, the solution will not be updated, and the next lower physical machines will be selected as a candidate machine to turn off. When multiple constraints are violated, the heuristic begins with the resource utilization that exceeds the allowed maximum the most, and migrates virtual machines utilizing more respective resource to other physical machines with lower total utilization.
4.5 Testbed and CoolCloud Implementation

4.5.1 Testbed Data Center Configuration

We have built a real virtualized data center testbed as shown in Figure. 4.3 to evaluate our design and ensure it can be practically applied to real world data centers. Currently we have four physical servers to host virtual machines, each configured with i7 3770 CPU, 16GB DDR3 memory. Each physical server is virtualized using VMware ESXi5 hypervisor. We have deployed 20 Ubuntu 12.04 LTS Linux virtual machines in this data center. Each VM is equipped with an iSCSI network storage and can be accessed by every physical server. A third server is used to host the heart of the data center vCenter, which manages all the VMs and hypervisors. vCenter is also responsible for sending out migration command once VM placement decisions are made. Two additional virtual servers are used to provide DNS, Active Directory Domain and network storage services. Figure. 4.4 is the topology of the data center configuration. The following provides detailed information about the hardware and software setup of the testbed:

- vCenter Server 5.0: Intel Core i7-3770@3.40GHz, 4 GB RAM, runs 64-bit Windows 2008 Server R2.
- vCenter Database: Intel Core i7-3770@3.40GHz, 4 GB RAM, runs 64-bit Windows 2008 Server R2 and Microsoft SQL Server 2005.
- ESX 5.0 Servers: Intel Core i7-3770@3.40GHz, 32 GB RAM.
- Network: 1Gbps vMotion network configured on a private LAN.
- Storage: 2 1TB iSCSI storage hosted by 2 Windows 2008 Server R2, shared by all the hosts.

We configure a Hadoop cluster built with Apache Hadoop 2.2.0 on top of our testbed data center to perform execution of MapReduce benchmarks. The cluster is composed of one master
node and 20 computing nodes running Ubuntu 12.04. Each node has 4 GB of memory and 40 GB hard disk. The Hadoop configuration uses the default settings and runs with Oracle JDK 1.7. The estimation of resource usage of a specific VM is based on the VM’s history resource usage as all applications have program phases [63] that last for a period of time. With the characterization of the workloads and benchmark suite used in this paper, the workload fluctuations range from seconds to minutes, which are actively monitored by the CoolCloud data center. In this design, we use one minute as the threshold for a stable program phase and the threshold for initiating VM migration/remapping.
4.5.2 Live Migration

Live VM migration has become available in recent years on modern virtualization platforms, e.g., VMware, Xen, which further enhances the efficiency and flexibility of data center management. Live VM migration is a technology that allows moving an active VM from one physical host to another without shutting down the VM. Through this process, users will only feel little or even no interruption of their service. The actual VM downtime (usually in milliseconds) during live migration depends on the implementation. This downtime is considered the cost of migration, which might affect system performance thus must be taken into account while designing dynamic VM placement algorithms [46]. For example, the pure stop and copy approach usually suffers long downtime (tens of seconds) since this approach suspends the VM at the beginning of migration. The entire memory contents and architectural state is then transferred to the destination host. After this, the VM is reinstantiated and the applications resume.

Another migration mechanism is called pre-copy. This approach reduces service downtime considerably by transferring memory contents to the destination host while the VM continues to execute on the source host. This is an iterative process. During the period when the active memory of the VM is transferred to the destination, the copy of the VM that is still executing will dirty some of the transferred pages by rewriting to them. The hypervisor memory management unit tracks the dirty pages, which are then resent to the destination in subsequent migration iterations. The iterative process continues until a very small working set size is reached or until a predefined iteration count limit is reached. At that point, the migration execution changes from the pre-copy to downtime phase, during which the VM is suspended and the remaining active memory of the VM and the architectural states, such as register contents, are transferred to the destination host. Since most of the memory of the VM has been transferred to the destination beforehand, the downtime is typically minimal (in tens of milliseconds). Today’s most popular commercial virtualization products, e.g., VMware, Xen, and Kernel Virtual Machine are all based on pre-copy-based live migration. In our CoolCloud im-
4.5.3 CoolCloud Software Design

CoolCloud is mainly implemented in Java with 5000 lines of code which is publicly available. The optimization model is implemented in 1100 lines of Java code excluding IBM CPLEX libraries. The heuristic is implemented in 1700 lines of Java code. And finally the VM controller is implemented using VMware vSphere 5 SDK suite with 2300 lines of Java code. The VM controller software design includes three major components and communicates with vCenter to manage the testbed data center. The first component is responsible of collecting runtime resource utilizations of each VM. These resources include CPU, memory, network and storage. This process is distributed to each VM where statistics are collected and sent to the data collector. The data collector then sends the VM statistics to the second component which is the optimization model mentioned in Section 4.3 that provides the optimal VM place-
ment solution. The third component is the VM migration commander which is responsible of sending out VM migration commands based on the placement solution from the optimization model. This component communicates with vCenter and places each VM onto the optimal server.

Migrating a VM from one PM to another PM requires extra CPU, memory and network bandwidth. We studied the live migration cost by migrating VMs to different servers in our data center testbed. Main observations are: 1. VM migration adds about an extra 10% to 20% CPU utilization; 2. the duration of migration mostly depends on the amount of VM memory size and network bandwidth. These observations suggest us to provide 10% CPU headroom for migration. The migration duration can be approximately modeled as: \[ \Delta t = 10 + 10 \times M(\text{GB}) \] (s) where \( \Delta t \) is the migration duration in seconds and \( M \) represents the VM’s memory size in GB.

4.5.4 VMware DRS and DPM

Distributed Resource Management (DRS) is designed by VMware to efficiently manage allocation of server resources to virtual machines in a cluster. As this is a commercial proprietary product, there is only limited information regarding the internal design. The core function of DRS is to provide dynamic load balancing of VMs across servers based on VMs’ resource needs (CPU and memory). DRS uses a greedy hill-climbing algorithm to check if a VM migration could reduce the cluster Imbalance Score \( I_c \), which is defined as the standard deviation of resource utilizations over all hosts. Migration would be applied if \( I_c \) can be reduced. This move-selection step is repeated until no additional beneficial moves remain or the current \( I_c \) is lower than the pre-defined \( I_c \). DRS is invoked periodically (by default, every 5 minutes) to calculate the current \( I_c \) and determine whether migration is necessary.

Distributed Power Management (DPM) is an extended feature of DRS aiming to save energy. DPM periodically checks resource utilizations of each ESXi server and recommends VM evacuation and powers off ESXi servers if the resource utilization is lower than a threshold.
Compared to VMware DRS and DPM solution, our design is more concise and integrated. The objective of our design is to minimize cluster energy consumption and the constraints themselves will provide the load balancing features. Our design pro-actively checks for optimization opportunities triggered by VM workload fluctuations or new VM placements. The server resources considered in our design are not limited by CPU and memory, we also consider network and storage utilizations, which eliminate potential bottlenecks caused by these resources.

4.6 Experiment Result

We first evaluate the energy conserving capability of our optimization framework (the ILP design) to demonstrate that optimal dynamic VM placement can be achieved. Secondly, we demonstrate that the heuristic design is capable of achieving near optimal results. Thirdly, we use simulation to demonstrate that the heuristic design can scale well to large scale server clusters.

In order to thoroughly examine whether the dynamic VM placement decisions could effectively result in a balanced and energy aware data center, long-running and fluctuating workloads are required to trigger the VM migration. These workloads include: Apache ab, Phoronix Test Suite [69] and HiBench [85]. HiBench is a widely-used benchmark suite for Hadoop provided by Intel to characterize the performance of MapReduce based data analysis running in data centers. While the benchmark programs are running, our dynamic VM placement software will keep monitoring the VMs and servers to make migration decisions when necessary. At the same time, we keep record of each physical server’s resource utilization and power consumption for a one hour period. We run all experiments three times and use the average as the result.

Besides testing CoolCloud in a Hadoop environment, we also show how CoolCloud performs when it comes to managing VMs hosting Docker containers. Docker is a lightweight
Docker container that helps software developers to build, ship and run distributed applications. It uses resource isolation features of the Linux kernel such as cgroups and kernel namespaces to allow independent containers to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines. Docker containers usually run in VMs and this technology is becoming very popular in regard of fast development and deployment of cloud applications.

For the evaluation of energy saving capabilities of VMware DRS [21] and our heuristic algorithm, the same testbed and workloads are used. We compare these three designs in regard of their abilities to balance workloads, server resources and their energy saving abilities. The results of network utilization, imbalance score and power consumption of each design are compared to demonstrate their overall performance.

The power consumption of each physical server is measured based on the work in [17, 62], where the full-system average power is approximately linear with respect to CPU utilization as given in eq. (4.10). It has proven to be an accurate way of measuring server power consumption especially in a data center environment where the total power consumption is an aggregation over a large number of servers.

\[ P_{Total} = P_{Dynamic} \cdot U_{Avg} + P_{Idle} \] (4.10)

In eq (4.10), \( P_{Total} \) is the total power consumption of the server, \( P_{Dynamic} \) is the dynamic power consumption of the CPU, \( U_{Avg} \) is the average CPU utilization and \( P_{Idle} \) is the power consumption when CPU is idle. In our experiment, all the metrics on the right side of eq.(4.10) are measured using the Intel Power Gadget [1].

4.6.1 Evaluation on Testbed

In the following experiment result charts, note that CoolCloud is the proposed optimization design, where CoolCloud(I) represents the ILP design and CoolCloud(H) represents the heuristic design.
Figure 4.6: Network Utilization in DRS

Figure 4.7: Network Utilization in CoolCloud
Figure. 4.6 shows the network utilization of each server while the testbed data center is managed by VMware DRS. As we can see there are big differences of network bandwidth consumption of each server. For example, within the 15 minutes examining period, server 1 only consumes less than 100 Kbps of bandwidth. Server 4 on the other hand, consumes more than 700 Kbps of bandwidth. This is because DRS does not balance the network resource utilizations across servers. This is especially harmful when several VMs that all require high network bandwidth are placed on the same server. This design flaw causes resource wastage: due to the bottle neck of one resource, other resources can not be fully utilized. For example, in the case of unbalanced network utilization, if a PM runs out of network bandwidth, even if it still has large amount of remaining CPU or memory resource, it is unlikely to accommodate any more VMs.

On the other hand, Figure. 4.7 shows the network utilization of each servers while the testbed data center is being managed by our dynamic VM placement design. To demonstrate the effectiveness of our design and to save space at the same time, we only show the result for CoolCloud ILP, since the result for Heuristic is similar. The network utilization starts unbalanced with server 4 having heavy network traffic (1200 Kbps) and server 2 having very little network traffic (0 Kbps). Our optimization model quickly detects this imbalance and provides the optimal placement solution. In about 3 minutes, the migrations are complete and the network bandwidth consumptions are balanced across all servers. This demonstrates our design solves the unbalanced issue in DRS, eliminating potential network bandwidth bottlenecks.

Figure. 4.8 shows the power consumption of the data center managed by No Migration, DRS, CoolCloud(I) and CoolCloud(H). Each case is monitored in a 60 minutes time period. The data center starts with the same workload and initial VM placement. The result shows both DRS and our design are capable of achieving power savings. DRS can provide 15.5% power savings on average compared to the settings where no management scheme is used at all. CoolCloud(I) and CoolCloud(H) achieved 28.6% and 28.3% power savings respectively when comparing with the case of no management scheme used, and this is over 15% gain of
power consumption measured here is the result of taking all costs including the cost of live migration into consideration.

Both DRS and our design provide power savings by turning off under utilized servers, however our design is capable of achieving the maximum power savings. This is because DRS mainly focuses on balancing CPU resource, and it only periodically (every 5 minutes) checks if any server is under utilized. This periodic checking may miss some energy saving opportunities due to the fluctuation of workloads. Further more, DRS does not provide the balancing of memory or network bandwidth across servers. This implies that some servers cannot be turned off due to resource wastage which leads to waste of energy. On the other hand, our design has an objective of minimizing energy consumption and all aspects of server resources are being considered. This creates a well balanced data center in regard of all resources, thus more servers can be turned off to achieve more energy savings. Notice that our design constantly monitors the server resource utilization in a pro-active fashion, thus responding quickly to the workload fluctuations and seizing every energy saving opportunities.

Figure. 4.9 provides the performance evaluation of CoolCloud in a Hadoop environment. In this experiment, we measure the execution time for four benchmark programs from Hi-
Figure 4.9: CoolCloud Performance Evaluation

Bench, i.e., WordCount, Sort, PageRank and Kmeans. For WordCount, the execution time is about the same across all three configurations, i.e., 132s for No Migration, 138s for CoolCloud and 130s for DRS. CoolCloud requires slightly longer time to complete execution due to the overhead of live migration. However for PageRank and Kmeans (825s for No Migration, 684s for CoolCloud and 756 for Kmeans), CoolCloud demonstrates significant lower execution time compared to No Migration and DRS. This is because No Migration can not resolve the resource contention issue experienced by VMs, and DRS only reacts to this issue every 5 minutes. On the other hand, CoolCloud is able to detect the resource contention proactively and respond quickly by initiating VM migration to resolve this issue. Note that the cost for live migration is fully considered in both the optimization model and the heuristic. The time duration for live migration typically ranges from several seconds to tens of seconds depending on the memory footprint of the VM. The small performance degradation comes from the live migration overhead and affects the applications performance running on that specific VM. CoolCloud prioritizes VMs’ with smaller memory footprints for migration thus significantly reduces the overall migration overhead.
Figure 4.10: Execution Time

Figure 4.11: Energy Consumption
Figure. 4.10 and Figure. 4.11 show the evaluation of CoolCloud managing VMs hosting Docker containers. For this configuration, there are 20 VMs in total and we deploy 5 Docker apache web servers in each VM. The goal of this experiment is to observe how CoolCloud performs when different number of requests are randomly sent to the Docker servers. We prepare 3 web server scripts that will send requests to each docker servers. The number of requests are randomly picked from 1K, 5K and 10K of requests. For example, Script 1 may send 10K requests to Docker server 5, 1K requests to Docker server 11, etc. Similarly, Script 2 may send 5K requests to Docker 5, 5K requests to Docker 11, etc. Figure. 4.10 provides the execution time of the 3 scripts under different management strategies. No Migration takes the longest time to complete in all three Scripts. In Script 2, No Migration takes 3780s to finish. We discovered the reason is that in Script 2, a large number of requests (140K) happened to be sent to the same server, resulting in high resource contentions among VMs residing on this server. This causes significant performance degradation when no migration strategies are used. However, CoolCloud is capable to quickly identify this SLA violation and moves VMs to servers with sufficient resource. DRS also improves performance by balancing resources among servers, however, network bandwidth still turns out to be a performance bottleneck. Figure. 4.11 gives the energy consumption of three configurations running three scripts. No Migration consumes the most energy due to resource contention among VMs. For Script 2, No Migration consumes 922KJ of energy due to one server taking significant longer time to finish the requests. Again, CoolCloud consumes 691KJ of energy executing Script 2. It responds quickly to resource contentions and identifies VMs that have completed their request tasks and consolidates these VMs to a smaller number of servers to save energy. On average, Cool Cloud achieves 20.7% and 13.9% energy savings compared to No Migration and DRS respectively.
4.6.2 Evaluation through Simulation

The evaluation result of the ILP and heuristic designs against the test bed data center has proven to provide better energy conservations compared to VMWare’s Dynamic Resource Planning design. In this section, we demonstrate that the heuristic design can be effectively applied to large-scale clusters to provide energy savings. To thoroughly evaluate the heuristic design, we designed an hybrid approach that combines profiling VM data from the test bed and simulating a large-scale cluster using the collected data.

4.6.2.1 VM Profiling

The goal of VM profiling is to generate large numbers of VMs with runtime information and feed these VMs’ runtime info as inputs to the ILP and heuristic to evaluate their performance in respect of energy saving capability and computation complexity. The VM profiling is accomplished while running benchmark programs on the 20 VMs in the data center testbed. To accelerate the profiling process and not lose the fluctuation of workloads, each profile lasts 30 minutes. The profile includes VM’s CPU, memory, network, harddisk utilization and power consumption footprint. Since each profile represents 20 VMs and requires 30 minutes to generate, we need 60 minutes to generate profiles for 40 VMs, 150 minutes for 100 VMs, 240 minutes for 160 VMs and so forth. In this paper, we generate profiles for 1000 VMs in total and provide the simulation result in the following.

4.6.2.2 Performance Comparison of CPLEX and Heuristic Algorithm

The simulation for the ILP design and the heuristic design are carried out with the same system configuration: A server with 2 Intel Xeon x5650 CPUs which has 24 virtual cores, Red Hat Enterprise Linux Workstation 6.6 (Santiago) with 2.6 kernel, and the total amount of memory is 47 GB. The optimization ILP formulation is solved by IBM CPLEX 12.5.

Figure. 4.12 displays the energy consumption result for the ILP design and the Heuristic design when the number of virtual machines in the data center ranges from 20 to 1000. In
Figure 4.12: Energy consumption for ILP and Heuristic

Figure 4.13: Computation time for ILP and Heuristic
the case of 1000 VMs, with the management of ILP, the data center energy consumption is 5280kJ and this number is 5401kJ for applying the heuristic design. This means the solution provided by the heuristic design only differs 2.3% from the optimal result. Overall, this result demonstrates that the heuristic design can provide solutions with only slight degradation on energy savings compared to the optimal ILP design.

Figure 4.13 displays the computation time of ILP and Heuristic. In the case of 20 VMs, ILP and Heuristic take comparable time for calculation with 180ms and 375ms respectively. At the point of 50 VMs, the computation time is about the same with 630ms and 661 respectively. However when there are more than 50 VMs, the computation time for solving ILP grows dramatically as the number of VMs increases. In the case of 1000 VMs, the computation time for ILP and Heuristic are 680s and 22s respectively. This result demonstrates that the heuristic design is highly computational efficient when it comes to large-scale clusters.

Overall, the simulation result shows that the heuristic design can provide near optimal solutions for energy savings and it is highly computational efficient making it a practical solution for large-scale data centers.

4.7 Conclusion

This paper presents a fine grained dynamic virtual machine placement framework to manage the mappings of VMs to physical servers. This framework solves the problem of finding the most energy efficient way (least resource wastage and least power consumption) of placing the VMs considering their resource requirements. We formulate the problem as a ILP problem which is the core of the framework. We study the opportunity of energy minimization in data center while meeting a set of constraints. However, finding the best placement policy is expensive. Therefore, a heuristic method is designed to obtain a near-optimal solution with significantly less computation time. The proposed framework design includes three major
components: runtime resource utilization collector, VM placement optimization model and the live migration commander.

What makes our design unique is that it is a complete solution that constantly monitors the workload changes over time in a data center and pro-actively provides VM placement solutions to guarantee performance and save energy. Our work considers all aspects of resources including CPU, memory, network and storage. It also takes into account of the cost of live VM migration which is often ignored in previous works. A real testbed data center implemented with industry product VMware vSphere 5 is used to evaluate the proposed framework. Experiment result demonstrates our design can effectively improve data center energy efficiency with each VM’s resource requirement satisfied. The heuristic design is proven to provide high scalability for large scale server clusters. Our design performs better than VMware’s Distributed Resource Scheduler (DRS) in respect of load balancing and power consumption.
CHAPTER 5. COOLCLOUD: OPTIMIZING ENERGY EFFICIENCY IN VIRTUALIZED DATA CENTERS

This paper is submitted to IEEE Transactions on Computers

Zhiming Zhang\textsuperscript{1}, Chan-Ching Hsu and J. Morris Chang\textsuperscript{2}

abstract

As cloud computing services continue to grow, managing energy consumption of data centers has become a great challenge. In this paper, we propose an optimization framework CoolCloud to minimize energy consumption in virtualized data centers. The proposed framework minimizes energy at two different layers: (1) minimize local server energy using dynamic voltage and frequency scaling (DVFS) exploiting runtime program phases. (2) minimize global cluster energy using dynamic mapping between virtual machines (VMs) and servers based on each VM’s resource requirement. Such optimization leads to the most economical way to operate an enterprise data center. On each local server, we develop a voltage and frequency scheduler (we call it the "cool scheduler") that can provide CPU energy savings exploiting applications’ run-time program phases. We provide a program phase prediction model based on time series analysis with the goal of managing each VM’s service level agreement. At the cluster level, we propose a practical solution for managing the mappings of VMs to physical servers. This framework solves the problem of finding the most energy efficient way (least resource wastage and least power consumption) of placing the VMs considering their resource

\textsuperscript{1}Primary researcher and author.

\textsuperscript{2}Author for correspondence.
requirements. A placement algorithm with low computation complexity is proposed to make our design scale well to the size of enterprise data centers. The CoolCloud framework is implemented with the most advanced hypervisor and management software stacks that represent the most recent data center infrastructures. Evaluation result against industry standard cloud computing benchmarks demonstrates that CoolCloud is capable of providing significant energy savings while maintaining service level agreements.

5.1 Introduction

In recent years, cloud computing services have become the fastest growing business around the world. The energy consumption continues to rise as more and more data centers are being built. In 2013, U.S. data centers consumed an estimated 91 billion kilowatt-hours of electricity [66], which is 2% of the total U.S. electricity consumption. This number is projected to increase to 140 billion kWh by 2020. How to design a more energy efficient data center has become one of the most important issues for cloud computing service providers.

As virtualization technology has matured, more and more cloud service providers are exploiting this technology to enhance efficiency and flexibility in data center management and resource provisioning. With the support of virtualization, resources of a single server can be divided into multiple isolated execution environments. This allows one physical machine to host several virtual machines (VMs), which helps to achieve higher per-server utilization that results in higher energy efficiency. A virtualized data center is comprised of many physical servers that host virtual machine templates which are used to run different applications and provide on demand computing environments to customers.

Virtualization does provide great flexibility for data center operators to optimize performance and energy efficiency, however it is a complicated task that requires careful design. Current research on managing virtualized data centers can be classified into three categories:

1. achieving the maximum performance under a given power budget. Research efforts focus
on how to allocate available power resources to servers [84, 39, 81] [83, 42]. 2. improving data center performance by increasing resource utilization and load balancing of VMs across physical servers [74, 90, 32, 18]. 3. achieving energy savings through VM consolidation algorithms [85, 24, 88]. In this paper, we focus on the last two problems that deal with data center performance and energy efficiency.

Improving energy efficiency in a virtualized data center can be achieved from two levels: the cluster level and the local server level. At the cluster level, we can monitor the resource utilization of all the VMs and servers. Intuitively, if a server hosts VMs with relatively low resource utilizations, these VMs can be consolidated to other servers that have enough resources remaining. When all the VMs have been evacuated from the aforementioned server, it can be turned into sleep mode to save energy. With the support of live migration, this process can be achieved with little interruption to user experience. At the local server level, we can monitor the CPU utilization of each server and adjust CPU frequency accordingly. In the case of high CPU utilization, frequency can be increased to enhance performance. On the other hand, when CPU utilization is low, frequency can be lowered to provide energy savings. Most of today’s processors support the technology of dynamic voltage and frequency scaling (DVFS), such as Intel’s SpeedStep, which makes DVFS energy aware design easy to deploy.

In this paper, we propose an optimization framework for large scale virtualized data centers around a set of energy conservation opportunities and service/resource constraints. We name our framework CoolCloud given its ability to cool down the data center that provides a more energy efficient cloud. CoolCloud optimizes energy consumption at both levels: the cluster level and the local server level. At the cluster level, a dynamic virtual machine placement algorithm is designed to find the most energy efficient way of placing VMs according to their resource requirement. Unlike previous works that only focus on CPU or memory utilization, the proposed algorithm actively monitors each VM and collects all of their runtime resource utilization which includes: CPU, memory, network and storage. The optimal placement solution will then be provided given the VM resource utilizations.
At the local server level, we provide a simple but effective energy aware DVFS design that adjusts CPU P-states exploiting the applications’ run-time program phases. We provide a mapping between memory access per instruction (MAPI) and CPU frequency. This mapping allows CoolCloud to select the optimal CPU frequency to operate servers. Unlike other research on DVFS designs that do not consider SLA, our design includes a prediction model based on time series analysis that uses autoregressive integrated moving average to forecast the next value of MAPI. The prediction model demonstrates much higher accuracy in forecasting the next MAPI value compared to last observation carry over which helps CoolCloud to maintain SLA while providing energy savings. In our SLA model, the SLA defines a task execution time constraint for the CPU. We assume the system performance is dominated by the CPU without considering the changing latency of I/O devices or network accesses. CoolCloud’s optimization at two levels are independent with each other, i.e., local server level needs not to communicate with cluster level to operate and provide energy savings, and vice versa.

We implement CoolCloud with Xenserver 6.5 and build a real data center testbed to evaluate our work. We select industry standard cloud computing benchmarks Hibench 2.6 and workloads that represent the most recent cloud computing applications. Experiment result demonstrates that CoolCloud can effectively provide significant energy savings while maintaining service level agreements. The main contributions of our work are:

- To the best of our knowledge, this work is the first to provide energy and performance optimizations at both the cluster level and the local server level.

- We provide a prototype implementation of a real testbed data center with Xenserver that is integrated with dynamic virtual machine placement and DVFS control.

- We propose a MAPI prediction model based on time series analysis that can accurately forecast the next value of MAPI which helps to maintain SLA.

- We provide a practical dynamic virtual machine placement algorithm that can be deployed in large scale data centers to find the most energy efficient way of placing VMs.
The remaining of the paper is organized in the following sequence. Design background is given in Section 5.2. We provide our CoolCloud framework design in Section 5.3. The implementation of our design is provided in section 5.4. Section 5.5 demonstrates the experiment results. Section 5.6 presents related work and Section 5.7 concludes this paper.

5.2 Design Background

Energy efficient design for data center requires a comprehensive understanding of the current state of the art architecture of data centers and available energy saving techniques. In this section, we provide some background on virtualized data center including the latest hypervisor technologies, data center management tools in the context of a Xen environment, live migration technologies, etc. We also introduce the advances in recent dynamic voltage and frequency scaling technologies and what to be mindful when using DVFS for energy saving. We present the motivations and challenges of using program phases to guide DVFS.

5.2.1 Virtualized Data Center

A virtualized data center contains thousands of servers, it takes advantage of the virtualization technology to create a pool of cloud infrastructure resources, i.e., cpu, memory, disk and network, etc. Typically, each physical server is installed with a Type-1 hypervisor which means the hypervisor runs directly on and interacts with the server hardware. Figure. 5.1 shows a typical server structure in a virtualized data center. As Xen is used in our later implementation, we describe this server in the context of Xen and Linux kernel. In Figure. 5.1, the physical machine is virtualized by the Xen hypervisor. Dom0 is the control domain that acts as an interface to Xen and provides access to the system hardware, creates, destroys and manages guest operating systems (VMs). Dom0 is essentially a Linux kernel that runs the Xen management tool stack and presents guest OSs with a set of common virtual hardware. On
Figure 5.1: Virtualized data center architecture
the other hand, DomU is called the user domain for guest operating systems. DomU is an unprivileged domain with no direct access to hardware or device drivers.

Nowadays, many cloud computing service providers are adopting virtualization technology for their cloud infrastructure. This is because virtualization can significantly improve IT agility, flexibility and scalability while offering cost savings. Workloads get deployed faster, performance and availability increases and operations become automated, resulting in IT that’s simpler to manage and less costly to own and operate. From a user point of view, virtual machines allow users to obtain administrative privileges if each user of the hosting resources is allocated a VM. This alleviates the task of system administrators and gives more control to application users [19]. It also provides performance isolation compared to traditional single OS servers where performance perturbation is introduced by simultaneous usage of multiple applications. In a virtualized data center, applications executed in one virtual machine would not be affected by applications running in another virtual machine.

When it comes to energy saving in a virtualized data center, the goal is to place VMs on to a smaller number of physical servers, so the remaining servers can turned into sleep mode. During this process, VMs need to be moved around the cluster given their resource requirement. This object can be achieved with the support of recent advances in live migration technology.

Live VM migration has become available in recent years on modern virtualization platforms, e.g., VMware, Xen, which further enhances the efficiency and flexibility of data center management. Live VM migration is a technology for moving an active VM from one physical host to another without shutting down the VM. Through this process, users will only feel little or even no interruption of their service. The actual VM downtime (usually in milliseconds) during live migration depends on the implementation. This downtime is considered the cost of migration, which might affect system performance thus must be taken into account while designing dynamic VM placement algorithms [46]. For example, the pure stop and copy mechanism usually suffers long downtime (tens of seconds) since this approach suspends the
VM at the beginning of migration. The entire memory contents and architectural states is then transferred to the destination host. After this, the VM is reinstantiated and the applications resume.

Another migration mechanism is called pre-copy. This approach reduces service downtime considerably by transferring memory contents to the destination host while the VM continues to execute on the source host. This is an iterative process. During the period when the active memory of the VM is transferred to the destination, the copy of the VM that is still executing will dirty some of the transferred pages by rewriting to them. The hypervisor memory management unit tracks the dirty pages, which are then resent to the destination in subsequent migration iterations. The iterative process continues until a very small working set size is reached or until an iteration count limit is reached. At that point, the migration execution changes from the precopy to downtime phase, during which the VM is stopped and the remaining active memory of the VM and the architectural states, such as register contents, are transferred to the destination host. Since most of the memory of the VM has been transferred to the destination beforehand, the downtime is typically minimal (in tens of milli-seconds). Today’s most popular commercial products, e.g., VMware, Xen, and Kernel Virtual Machine are all based on pre-copy-based live migration.

5.2.2 Dynamic Voltage and Frequency scaling

In general, processor frequency is a key metric of system performance and higher frequency generally means better overall system response or throughput. However high operating frequency may also lead to high potential of energy waste especially in data centers since cloud computing services usually contain many I/O and memory transactions. Our research attempts to minimize the energy waste caused by memory-related stall cycles by using the technique of dynamic voltage frequency scaling (DVFS). DVFS has been widely used to provide energy efficient designs. Most modern CPUs supports DVFS to optimize performance and energy
efficiency. For example, Intel’s SpeedStep technology adjusts CPU frequency based on CPU utilization.

More fine grained designs with DVFS monitor program’s execution characteristic and adjust CPU frequency accordingly. For example, memory access pattern of a program can be used as an index to adjust CPU frequency. Due to the speed gap between CPU and main memory, CPU needs to wait for memory transaction to finish before continuing execution which results in energy waste. One ideal solution is to minimize the CPU frequency every time when the CPU is stalled by main memory access, and then switch back to high frequency after the stall is over. In this case, energy can be saved with no performance degradation. However in practice, CPU will be unavailable for about 50 $\mu$s to 650 $\mu$s [48, 2] during a DVFS operation. This time-span is much larger than the main memory latency which is around 100 nano seconds. Thus DVFS can not be applied every time the processor is stalled by a main memory access.

A practical solution is to exploit the program phases (i.e. memory intensive phase and CPU intensive phase [38], [72]). The memory intensive phase is the time duration when the program has many memory activities. We can turn down the processor frequency during this time period to save energy but still achieve comparable performance. The CPU intensive phase is the time duration when most of the work is done on the CPU. The CPU should run on high frequency during this time period to guarantee performance. We call the memory intensive phase and the CPU intensive phase two distinct "program phases".

We use a simple motivation example to illustrate this concept. Figure. 5.2 is the execution of mcf (a benchmark program from SPEC CPU2006 used for single-depot vehicle scheduling in public mass transportation) on two different frequencies. We examine its memory access per instruction (MAPI) value when CPU is running on 1.998 GHz and 2.664 GHz respectively. MAPI is the number of Memory Access Per Instruction which can be used as an indicator of a program’s memory access intensiveness. Observation shows two distinct phases: memory intensive phase ($MAPI > 0.008$) and computation intensive phase ($MAPI < 0.006$). The
Figure 5.2: Execution behavior of $mcf$ on 1.998 GHz and 2.664 GHz
execution time for memory intensive phases is about the same no matter when the program is running on 1.998 GHz or 2.664 GHz. However, CPU running on 1.998 GHz causes the execution time of computation intensive phases obviously longer than when the CPU is running on 2.664 GHz.

This example demonstrates that performance drops when CPU frequency is reduced. However, for the same amount of frequency drop, the performance degradation depends on the program phases. This implies performance suffers less degradation at memory intensive phase for the same amount of frequency drop. This motivates us to switch down the frequency during memory intensive phases to save energy without much performance degradation. How to select the optimal frequency given a specific program phase will be discussed in Section 5.3.
5.3 CoolCloud Framework Design

CoolCloud optimizes energy consumption of the data center at two levels: local server level and the cluster level. Figure 5.3 is the high level design of CoolCloud. At the local server level, VMs with different workloads are hosted on the hypervisor, we monitor the accumulated workload program behavior through CPU performance counters and calculate metrics including MAPI, reorder buffer stall cycles, halted cycles, etc. We can identify the program phase through these collected metrics and select an optimal frequency to execute the workload in order to improve energy efficiency.

At the cluster level, the optimization goes through three major steps. The first step is responsible of collecting runtime resource utilizations of each VM. These resources include CPU, memory, network and storage. The second step is to calculate the optimal VM placement solution with an computation efficient algorithm. The objective of the algorithm is to minimize data center energy consumption without affecting each VM’s performance. The model takes each VM’s resource requirements as its constraints to guarantee performance. The final step is a commander responsible of sending out VM migration commands based on the placement solution generated from the placement algorithm. Live migration [46] is used here so service of each VM will not be interrupted during dynamic placements. Our design also considers the VM migration cost which is often ignored in past works. In the following, we provide the design details of optimizations at both levels, e.g., how to find the optimal frequency according to collected metrics, how to maintain service level agreement and details of the placement algorithm, etc.

5.3.1 Local Server Level Optimization

The energy saving capability of DVFS strategies depends on the SLA and the amount of memory accesses in a program. CPU frequency must be carefully chosen based on the distribution of the memory accesses. In order to find out the mapping between a specific program
phase and optimal frequency under a certain SLA requirement, we define program phases and a model that can help us calculate the desired running frequency. The program phases are mainly determined by three statistics captured at run-time using performance monitors: $MAPI$, $CPI_{exe}$, $h(f)$. We first give definitions for the behavior statistics we use.

- **MAPI**: Memory Access Per Instruction, to determine the memory access intensiveness of a thread [35].

$$MAPI = \frac{Bus_{Trans\_Mem}}{Instr\_Exe}$$ \hspace{1cm} (5.1)

where $Bus_{Trans\_Mem}$ is the number of main memory accesses and $Instr\_Exe$ is the number of instructions executed.

- **$CPI_{exe}$**: cycle per instruction when CPU pipeline not stalled by memory transactions.

- **$IC(f)$**: instruction count, total number of instruction executed in one second at CPU frequency $f$.

- **$\Delta m$**: latency of the main memory.

- **$h(f)$**: number of cycles when CPU is halted while operating at frequency $f$.

- **$o(f)$**: stall cycles caused by reasons other than memory access while CPU running at $f$.

- **$\alpha$**: memory latency overlap rate. This factor represents the out-of-order execution before CPU gets stalled by a memory access.

The cycle usage for a CPU operating on frequency $f$ within a second can be expressed as:

$$f = IC(f) \times CPI_{exe} + \alpha \times MAPI \times IC(f) \times \Delta m \times f$$

$$+ h(f) + o(f)$$ \hspace{1cm} (5.2)

where $IC(f) \times CPI_{exe}$ is the number of cycles while the CPU is not stalled by memory transactions neither halted. $\alpha \times MAPI \times IC(f) \times \Delta m \times f$ is the number of stall cycles due to main memory access. Notice that quantities on both sides of eq. (5.2) are in cycles/sec.
$h(f)$ represents the number of cycles when CPU is halted while operating at frequency $f$. The CPU gets halted when there is no work to be done, the CPU starts running an idle thread (HLT instructions) and enters its idle state. CPU stall happens when the CPU is still executing program instructions but waiting for the operand or data (usually because of the latency of memory) to be available.

In recently published models [11, 12], the authors ignore the effect of out-of-order execution and memory level parallelism in superscaler processors which leads to prediction errors. In our model we define $\alpha$ to represent this effect and enhance the accuracy of our model. The value of $\alpha$ is determined by the processor issue rate, re-order buffer size and system memory latency. In general, most of the stall cycles are caused by main memory access, thus we can ignore the $o(f)$ (e.g. L1 cache miss and branch miss prediction related stalls) in eq. (5.2) with little impact on the accuracy of eq. (5.2). eq. (5.2) is rewritten into:

$$f \approx IC(f) \times CPI_{exe}$$

$$+ \alpha \times MAPI \times IC(f) \times \Delta m \times f + h(f)$$

The instruction count IC(f) can be derived from eq. (5.3):

$$IC(f) \approx \frac{f - h(f)}{\alpha \times MAPI \times \Delta m \times f + CPI_{exe}}$$

In our performance model, we consider instruction count in a given interval of time as the performance measurement. Thus, the performance loss for CPU running on frequency $f$ compared to CPU running on the highest frequency $F$ can be defined as:

$$\delta = \frac{IC(F) - IC(f)}{IC(F)}$$

When the SLA requirement is given as a percentage of the maximum system performance, the required performance loss can be calculated as:

$$\delta = 1 - SLA$$

Let $f^{n-1}$ be the frequency level of a thread $t$’s $(n - 1)$th execution, its program behaviors $CPI_{exe}, MAPI, h(f)$ are monitored by the CPU during $t$’s $(n - 1)$th execution. Based on
our experiment, we have discovered that the number of halted cycles \( h(f) \) depends on CPU frequency which can be expressed in the following equation:

\[
h(f_1) \approx \frac{f_1}{f_2} \times h(f_2)
\]

(5.7)

where \( f_1 \) and \( f_2 \) are two different CPU frequencies. After collecting all the program behavior statistics, combine eq. (5.4), (5.5), (5.7) and obtain eq. (5.8), which is the equation that provides the desired operating frequency for \( t \)'s \( n \)th execution:

\[
f_{\text{target}}^n = \frac{IC(F) \times (1 - \delta) \times CPI_{\text{exe}}}{1 - IC(F)(1 - \delta) \alpha \times MAPI \times \Delta m - \frac{h(f_{n-1}^n)}{f_{n-1}^n}}
\]

(5.8)

where \( IC(F) \) can be derived from eq. (5.4).

\[
IC(F) \approx \frac{F - \frac{F - f_{n-1}^n h(f_{n-1}^n)}{\alpha \times MAPI \times \Delta m \times F + CPI_{\text{exe}}}}
\]

(5.9)

Eq. (5.8) is our proposed model that provides the desired operating frequency \( f_{\text{target}} \) given its program phases and SLA requirement (i.e. target performance loss \( \delta \)).

### 5.3.2 MAPI prediction model

Now we have defined the relationship between desired frequency and program phases. In order to execute the program on a desired frequency, we need to know the MAPI value first which requires an effective prediction model. This model is also important for maintaining SLA as if the MAPI is predicted wrong, the program will be executing on an undesired frequency. From Figure. 5.2 we can see the MAPI fluctuation within a time span is not significant, thus one can simply use the last observation value carry over approach to predict the next MAPI value without introducing significant prediction errors. However, as shown in Figure. 5.4 there are also many programs that exhibit significant variance in the MAPI value over a short time span. For such programs, the simple last observation value carry over approach will introduce significant prediction errors and may cause SLA violations.

Figure. 5.4 is essentially a MAPI time series, thus we try to use time series analysis to build models for predicting future MAPI values. We define \( \{Y_t\} \) as the value of MAPI over time
Figure 5.4: MAPI data: Hibench Kmeans

t = 1, 2, \ldots, T in the scale of a second, our goal is to forecast the next MAPI value $Y_{T+1}$. Time series analysis includes four components to characterize a given data set: 1. Trend component, which is a persistent, overall upward or downward pattern; 2. Cyclical component, which is the repeating up and down movements; 3. Seasonal component, which is a regular pattern of up and down fluctuations; 4. Irregular component, which describes erratic, unsystematic, residual fluctuations, due to random variation or unforeseen events. They present short durations and are non-repeating. As shown in Figure. 5.4 the selected MAPI data does not show an overall trend overtime. However, it does show some cycles in small duration which are reflected by the up and down swings that vary in length. Seasonal component is not observed and the MAPI data contains many irregular patterns.

We choose the autoregressive integrated moving average (ARIMA) model for forecasting, which is a general class of models for forecasting a time series. Given the characteristics of MAPI that demonstrate no obvious trends, we can treat it as a stationary series. The ARIMA forecasting equation for a stationary time series is a linear (i.e., regression-type) equation in
which the predictors consist of lags of the dependent variable and/or lags of the forecast errors.

As the MAPI data does not present much seasonal feature, we can select the non-seasonal
ARIMA equation, i.e., the ARIMA\((p, d, q)\) model, where:

- \(p\) is the number of autoregressive terms,
- \(d\) is the number of non-seasonal differences needed for stationarity,
- \(q\) is the number of lagged forecast errors in the prediction equation.

The forecasting equation is constructed as follows. First, let \(y\) denote the \(d\)-th difference of \(Y\), which means:

- If \(d = 0\), \(y_t = Y_t\).
- If \(d = 1\), \(y_t = Y_t - Y_{t-1}\).
- If \(d = 2\), \(y_t = (Y_t - Y_{t-1}) - (Y_{t-1} - Y_{t-2}) = Y_t - 2Y_{t-1} + Y_{t-2}\).

In terms of \(y\), the general forecasting equation is:

\[
\hat{y}_t = \mu + \phi_1 y_{t-1} + \cdots + \phi_p y_{t-p} - \theta_1 e_{t-1} - \cdots - \theta_q e_{t-q}.
\]

To identify the appropriate ARIMA model for \(Y\), we test the model by choosing different
\((p, d, q)\) values in order to determine the order of differencing \((d)\) needed to stationarize the
series, the number of autoregressive terms \((p)\), and the number of moving average terms \((q)\).

With the analysis above, we choose \(p = 0\), \(d = 1\) and \(q = 1\) and fit ARIMA\((0, 1, 1)\) with the
MAPI data. Let \(y_t = Y_t - Y_{t-1}\) and \(Y_t\) be the MAPI value after log transformation. Based on
this model, we forecast for the future MAPI value.

As demonstrated in Figure. 5.5, ARIMA can effectively forecast the future MAPI values
which helps us to apply the correct CPU frequency for energy saving and maintaining SLA.
As the training sample size affects the prediction accuracy and time consumption, we need to
study what sample size can provide desired accuracy while staying computation efficient. In
the following, we choose 100, 50 and 10 as sample size to evaluate which sample size can achieve desired prediction accuracy while meeting the time constraints. We use \( n_{\text{train}} \) data points for fitting ARIMA\((0, 1, 1)\) and forecast the next point (the testing point). Denote the forecast as \( \hat{Y} \) and the actual value as \( Y \). To measure the forecasting accuracy, we use the Mean Squared Error (MSE) between forecasted values and actual values:

\[
MSE = \frac{1}{n_{\text{test}}} \sum_{i=1}^{n_{\text{test}}} (\hat{Y}_i - Y_i)^2,
\]

where \( n_{\text{test}} \) is the number of the data used for testing, \( \hat{Y}_i \) is the forecast, and \( Y_i \) is the actual observation. \( MSE \) measures the average performance of forecast.

Table 5.1: Forecast accuracy as measured by the Mean Squared Error for 5 MAPI time series

<table>
<thead>
<tr>
<th>( n_{\text{train}} )</th>
<th>100</th>
<th>50</th>
<th>10</th>
<th>locf</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAPI 1</td>
<td>4.17e-08</td>
<td>4.24e-08</td>
<td>4.70e-08</td>
<td>5.60e-08</td>
</tr>
<tr>
<td>MAPI 2</td>
<td>1.43e-04</td>
<td>1.45e-04</td>
<td>1.69e-04</td>
<td>2.67e-04</td>
</tr>
<tr>
<td>MAPI 3</td>
<td>1.61e-04</td>
<td>1.76e-04</td>
<td>2.06e-04</td>
<td>2.31e-04</td>
</tr>
<tr>
<td>MAPI 4</td>
<td>1.93e-04</td>
<td>1.96e-04</td>
<td>5.82e-04</td>
<td>2.85e-04</td>
</tr>
<tr>
<td>MAPI 5</td>
<td>4.65e-03</td>
<td>4.66e-03</td>
<td>1.34e-02</td>
<td>9.42e-03</td>
</tr>
</tbody>
</table>
Table 5.1 gives the accuracy under different sample sizes. Sample size 100 provides the best accuracy over all five benchmarks with average computation time as 11ms, sample size 50 provides close accuracy to sample size 100 with computation time as 5ms. Sample size 10 provides the worst prediction accuracy although it can generate results in 2ms. Overall, we decide to use 50 as sample size since it can provide desired accuracy with the least computation time.

5.3.3 Cluster Level Optimization

At the cluster level, we optimize the energy consumption by monitoring each VM’s resource utilization and dynamically consolidate VMs on to the least number of physical servers. The dynamic placement process is mindful of each VM’s resource requirement. We first define the notations we use before introducing the placement algorithm. Table 5.2 provides the definitions of the symbols we use in the algorithm. \( N_{VM} = \{ VM_0, \ldots, VM_{M-1} \} \) is the set of virtual machines with cardinality \( |N_{VM}| = M \). \( N_{PM} = \{ PM_0, \ldots, PM_{N-1} \} \) is the set of physical machines with cardinality \( |N_{PM}| = N \). The problem is defined as follows:

Given (1) the power and time requirements of virtual machine \( m \) to run on physical machines \( n \), \( P_{mn} \) and \( T_{mn} \), (2) the migration cost of each virtual machine in power, \( P_{mn}^{migrate} \) and in time, \( T_{mn}^{migrate} \), (3) the physical machine capacity in terms of cpu, memory, hardware and bandwidth utilization, \( H_{n}^{CPU}, H_{n}^{MEM}, H_{n}^{HD} \) and \( H_{n}^{BW} \), (4) the virtual machine utilization requirements for CPU, \( U_{m}^{CPU} \), memory, \( U_{m}^{MEM} \), hard disk, \( U_{m}^{HD} \) and bandwidth, \( U_{m}^{BW} \) (5) the active and sleep mode power use of the physical machines, \( P_{n}^{active} \) and \( P_{n}^{sleep} \), the problem is to minimize the system energy consumption, denoted as \( E \), by placing virtual machines on physical machines PMs, deploying VMs in active mode with the consideration of the migration cost. The output of the algorithm includes the virtual machine destinations, migration indicators and operation mode of physical machines specified as, \( l_{mn}, g_{mn} \) and \( o_{n} \).
Table 5.2: Definitions of Important Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Number of physical machines to serve virtual machines</td>
</tr>
<tr>
<td>$M$</td>
<td>Number of virtual machines</td>
</tr>
<tr>
<td>$P_{\text{active}}$</td>
<td>Basic power level of physical machines in active mode</td>
</tr>
<tr>
<td>$P_{\text{sleep}}$</td>
<td>Power level of physical machines in sleep mode</td>
</tr>
<tr>
<td>$\text{Period}$</td>
<td>Time period for which the solution pertains</td>
</tr>
<tr>
<td>$P_{\text{migrate}}_{mn}$</td>
<td>Power level for VM $m$ migrating to PM $n$</td>
</tr>
<tr>
<td>$T_{\text{migrate}}_{mn}$</td>
<td>Time for VM $m$ migrating to PM $n$</td>
</tr>
<tr>
<td>$H_{\text{CPU}}$</td>
<td>Limit on CPU utilization of physical machines</td>
</tr>
<tr>
<td>$H_{\text{MEM}}$</td>
<td>Limit on memory utilization of physical machines</td>
</tr>
<tr>
<td>$H_{\text{HD}}$</td>
<td>Limit on hard disk utilization of physical machines</td>
</tr>
<tr>
<td>$H_{\text{BW}}$</td>
<td>Limit on network bandwidth utilization of physical machines</td>
</tr>
<tr>
<td>$N_{\text{VM}}$</td>
<td>Set of VMs, $</td>
</tr>
<tr>
<td>$N_{\text{PM}}$</td>
<td>Set of PMs, $</td>
</tr>
<tr>
<td>$U_{\text{CPU}}$</td>
<td>Virtual machine CPU utilization, $U_{\text{CPU}} = {U_{\text{CPU}}^m, \forall m \in N_{\text{VM}}}$</td>
</tr>
<tr>
<td>$U_{\text{MEM}}$</td>
<td>Virtual machine memory utilization, $U_{\text{MEM}} = {U_{\text{MEM}}^m, \forall m \in N_{\text{VM}}}$</td>
</tr>
<tr>
<td>$U_{\text{HD}}$</td>
<td>Virtual machine hard disk utilization, $U_{\text{HD}} = {U_{\text{HD}}^m, \forall m \in N_{\text{VM}}}$</td>
</tr>
<tr>
<td>$U_{\text{BW}}$</td>
<td>Virtual machine network bandwidth utilization, $U_{\text{BW}} = {U_{\text{BW}}^m, \forall m \in N_{\text{VM}}}$</td>
</tr>
<tr>
<td>$\mathcal{E}$</td>
<td>Total system energy consumed</td>
</tr>
<tr>
<td>$L$</td>
<td>Placement matrix (decision variable), $L = (l_{mn})_{M \times N}$</td>
</tr>
<tr>
<td>$G$</td>
<td>Migration matrix (decision variable), $G = (g_{mn})_{M \times N}$</td>
</tr>
<tr>
<td>$O$</td>
<td>Operation mode vector (decision variable), $O = (o_n)_{1 \times N}$</td>
</tr>
</tbody>
</table>

5.3.3.1 Energy Model

The objective of the algorithm is to minimize the energy consumption of the data center which is defined in the following:

$$
\min \mathcal{E} = \sum_{m \in N_{\text{VM}}, n \in N_{\text{PM}}} (P_{mn} \cdot \text{Period} \cdot l_{mn}) \\
+ \sum_{m \in N_{\text{VM}}, n \in N_{\text{PM}}} (P_{\text{migrate}}_{mn} \cdot T_{\text{migrate}}_{mn} \cdot g_{mn}) \\
+ \sum_{n \in N_{\text{PM}}} (P_{\text{active}}^n \cdot \text{Period} \cdot o_n) \\
+ \sum_{n \in N_{\text{PM}}} [P_{\text{sleep}}^n \cdot \text{Period} \cdot (1 - o_n)].
$$

(5.10)
Eq. (5.10) is the energy consumption as a summation of the virtual machine execution energy, migration energy, active physical machine energy and sleep physical machine energy [45]. The power consumption of each VM is related to its resource utilization, i.e., CPU, memory, etc., which can be modeled as:

$$P_{vm} = P_{cpu} + P_{mem} + P_{disk} + P_{static} + P_{other}$$ (5.11)

In eq. (5.11), $P_{mem}$ is the memory power consumption, $P_{disk}$ is the hard disk power consumption, $P_{static}$ represents the idle system power that stays constant, and $P_{other}$ is the power consumption of other components of the server. Since CPU is the major consumer of energy in a server, while memory, disk usually consume little energy compared to CPU, and their energy consumption does not fluctuate much, we assume it to be constant in our energy model. Thus we rewrite eq. (5.11) into eq. (5.12):

$$P_{vm} = P_{cpu} + P_{other}$$ (5.12)

$P_{cpu}$ is the cpu power consumption and $P_{other}$ represents power consumption of all other components that remain stable and constant. Since at the local server level, CPU’s P-state is being modified according to program phases, which means its power consumption will also change over time. This affects the VM power consumption, and in order to accurately estimate the power consumption of a VM, we need to update the most recent CPU P-state from local server DVFS controller to cluster level controller. The power consumption of CPU can be modeled as:

$$P_{cpu} = \alpha_{cpu} u_{cpu} + \gamma_{cpu}$$ (5.13)

where $\alpha_{cpu}$ is a power parameter determined by the CPU P-state, $u_{cpu}$ denotes the CPU utilization and $\gamma_{cpu}$ is a CPU model specific constant determined by CPU idle power[54]. We will provide the value of all the parameters of the CPU we use for our implementation in Section 5.4. With eq. (5.13), we can easily calculate the VM power consumption given the CPU utilization and P-states.
5.3.3.2 Dynamic VM Placement Algorithm

In the following, we present a computation efficient VM placement algorithm that scales to the size of an enterprise data center with thousands of VMs in a cluster. The objective of the algorithm is to minimize energy consumption at the cluster level, however, because the number of nodes (i.e., physical and virtual machines) changes and resource requirements vary (i.e., CPU, memory, hard disk, and network bandwidth) frequently, the algorithm needs to respond fast to the dynamics, thus an algorithm with low computation complexity is required for our design to be implemented for solving the proposed optimization model in practice. The pseudocode is shown in Algorithm 6 and Algorithm 7, and is implemented using Java programming language. It takes the collected resource utilization of each VM as input; the algorithm decides the placement of virtual machine, virtual machine migration, and operation mode of physical machine. In the following, we first discuss the algorithm design principles and then the details about the algorithm.

Two stages are devised in the algorithm with separate objectives. In the first stage, we want to determine whether a feasible solution exists prior to performing virtual machine consolidation, which is the second stage. Since the model specifies the maximum resource utilization of physical machines in each category (i.e., resource constraints), the algorithm checks if the current operation (i.e., \( l'_{mn} \)) violates the constraints. If a constraint is violated, the algorithm starts to find a feasible solution by performing virtual migration. The objective in this stage is to obtain a feasible solution as an initial solution to begin virtual machine consolidation. If no such feasible solution can be found, the algorithm determines no consolidation is necessary, and an alternative will be adopted when the problem is infeasible. In the following stage, the objective is looking for a better virtual machine placement decision in terms of energy consumption. Considering the difference between the operation energy in active and sleep mode of physical machines, switching physical machines into sleep mode results in reducing energy consumption. With such idea, placing virtual machines to a less number of physical machines is necessary to produce a better solution. Therefore, the attempt is to reach a new solution with
Algorithm 6 Energy-saving VM Placement

\textbf{Input:} Period, $\mathcal{L}'$, $N_{VM}$, $N_{PM}$, $P$, $P_{active}$, $P_{sleep}$, $U_{CPU}$, $U_{MEM}$, $U_{HD}$, $U_{BW}$, $T_{migrate}$, $H_{CPU}$, $H_{MEM}$, $H_{HD}$, $H_{BW}$ and $P_{migrate}$.

\textbf{Output:} $\mathcal{L}$, $\mathcal{G}$ and $\mathcal{O}$.

1: Create resource utilization monitor;
2: $\mathcal{L} = \mathcal{L}'$, $\mathcal{G} = 0$, and given $\mathcal{O}$;

\textbf{STAGE 1: Feasible Solution Initialization}
3: \textbf{while} The solution is not feasible \textbf{do}
4: \hspace{1em} \textbf{for} PMs with high utilization \textbf{do}
5: \hspace{2em} \textbf{if} The constraint is violated \textbf{then}
6: \hspace{3em} \textbf{for} VMs utilizing more resources \textbf{do}
7: \hspace{4em} Migrate VMs to PMs with lower utilization if constraints met;
8: \hspace{3em} \textbf{end for}
9: \hspace{2em} \textbf{if} The constraint still not satisfied \textbf{then}
10: \hspace{3em} \textbf{for} VMs utilizing more resources \textbf{do}
11: \hspace{4em} \textbf{for} PMs with lower utilization \textbf{do}
12: \hspace{5em} Exchange VMs if constraints satisfied;
13: \hspace{4em} \textbf{end for}
14: \hspace{3em} \textbf{end for}
15: \hspace{2em} \textbf{end if}
16: \hspace{1em} \textbf{end if}
17: \textbf{end for}
18: \textbf{end while}
19: \textbf{if} A feasible solution not found \textbf{then}
20: \hspace{1em} Adopt alternative for operation and leave;
21: \textbf{end if}

an improved energy consumption by consolidating virtual machines to a smaller number of physical machines.

The algorithm is designed to be executed whenever the context changes, for example, virtual machines are done with its tasks, or more virtual machines are requested to perform more tasks, or in cases where input values have to be updated and the need of producing new solutions. However, the computation overhead might appear to be high, if the algorithm needs to perform constantly. To lower such overhead, Period can be used to control the time inter-
Algorithm 7 Energy-saving VM Placement (Continue Stage 1)

STAGE 2: Virtual Machine Consolidation

1: Create energy cost monitor;
2: repeat
3: for Active PMs with lower total utilization do
4: for Another active PMs with lower total utilization do
5: Migrate all VMs on first PM to second PM;
6: if Constraints violated then
7: for Violated constraints with higher utilization do
8: for VMs utilizing more resources do
9: for Another active PMs with lower total utilization do
10: Migrate VMs from second PM to third PM if not exceed the maximum;
11: end for
12: end for
13: end for
14: end if
15: if Constraints violated then
16: No solution updated;
17: end if
18: end for
19: end for
20: until The solution not improved.
21: return ℒ, ℓ, and Ω.

val to run the heuristic, instead of producing solutions pro-actively. Overall, the computation complexity of the algorithm is \( O(MN(\log M)^2 \log N) \) in the worst case.

Before going into the first-stage of the algorithm, we create resource utilization monitors in order to keep track of the resource utilization status of physical machines. As the algorithm goes along, these monitors are updated whenever a better solution is found. The utilization level is calculated based on the cumulative resource allocated to virtual machines on each physical machine, namely, \( U^{CPU} \), \( U^{MEM} \), \( U^{HD} \) and \( U^{BW} \). The current operation strategy is taken as the initial solution, i.e., \( ℒ = ℒ′ \), \( ℓ = 0 \), and \( Ω \) is given. The algorithm compares the monitored resource utilization levels with the maximum resource utilization values,
i.e., $H_{CPU}$, $H_{MEM}$, $H_{HD}$ and $H_{BW}$. If any of constraints is not satisfied, the solution is not feasible, and a feasible solution needs to be found before entering the next stage of the algorithm.

In order to find a feasible solution, the heuristic tries migrating virtual machines virtually with the goal of reducing physical machine resource utilization. It first puts together and sorts utilization level in all categories of physical machines. Starting from the physical machine that consumes the highest utilized resource which exceeds the capacity (can be CPU, memory, hard dish or bandwidth), and on that physical machine we try two methods (line 4-7 and 8-11, respectively). First, we sort through each physical machine to find out which virtual machines utilize more respective resource, and find destination physical machines with lower utilization level respectively. The algorithm then calculates the new utilization levels for physical machines involved in migration, e.g., source and destination physical machines. If after the above migration attempts, the constraint still cannot be satisfied, the heuristic will try the following migration strategy. Similarly, starting with virtual machines that utilize more respective resources, we search for physical machines with a lower resource utilization level. Then virtual machines on the selected physical machine will be picked for migration. The algorithm intends to migrate virtual machines with higher resource utilization so that the opportunity of finding a feasible solution with less migration cost is higher. As the resource monitor keeps track of utilization levels, as long as all constraints are satisfied, the program can move into the second stage. However, if no feasible solution could be found after the procedure in the first stage, we determine there is no feasible solution to work on for energy saving procedure, and leave the routine.

The second stage of the heuristic serves the purpose of consolidating virtual machines in order to sleep physical machines, reducing energy consumption. Before performing the energy-saving procedure, we create the energy cost monitor to keep track of energy consumption of physical machines, migration cost and entire system status. In addition to respective resource utilizations, the total resource utilization of physical machines is summed up. Starting
from physical machines having lower total utilization (first physical machine in line 17, second in 18), the algorithm migrates virtual machines on the first physical machine to the second one, in order to switch off the first. However, if a constraint is violated after this migration, virtual machines will be migrated to other physical machines until constraints are satisfied, or until all other physical machines have been tried with no feasible solution reached. In the latter case, the solution will not be updated, and the next lower physical machines will be selected as a candidate machine to turn off. When multiple constraints are violated, the heuristic begins with the resource utilization that exceeds the allowed maximum the most, and migrates virtual machines utilizing more respective resource to other physical machines with lower total utilization.

5.4 Implementation of CoolCloud

In order to thoroughly evaluate our design, we build a fully capable testbed data center with Xenserver and its management tool stacks with the CoolCloud design implemented into the data center. Figure 5.6 provides the overall structure of the data center with the CoolCloud framework. Currently we have four physical servers to host virtual machines, each configured with i7 3770 CPU, 16GB DDR3 memory. Each physical server is virtualized using Xenserver 6.5 hypervisor. We have deployed 20 Ubuntu 12.04 LTS Linux virtual machines in this data center. We configure an iSCSI network storage server that can be accessed by every physical server. All the VMs can live migrate to any server freely. A third server is used to host the heart of the data center XenCenter, which manages all the VMs and hypervisors. XenCenter is also responsible for sending out migration command once VM placement decisions are made. Two additional virtual servers are used to provide DNS, Active Directory Domain and network storage services. The following provides detailed information about the hardware and software setup of the testbed:
Figure 5.6: CoolCloud Implementation Architecture
- XenCenter Server 6.5: Intel Core i7-3770@3.40GHz, 4 GB RAM, runs 64-bit Windows 2008 Server R2.

- XenCenter Database: Intel Core i7-3770@3.40GHz, 4 GB RAM, runs 64-bit Windows 2008 Server R2 and Microsoft SQL Server 2005.

- XenServer: Intel Core i7-3770@3.40GHz, 32 GB RAM.

- Network: 1Gbps vMotion network configured on a private LAN.

- Storage: 2 1TB iSCSI storage hosted by 2 Windows 2008 Server R2, shared by all the hosts.

5.4.1 Local Server Level

The DVFS controller monitors the program phases and adjusts CPU P-states accordingly. XenServer is based on a standard Linux distribution, but for performance, maintainability, and compatibility reasons ad-hoc modifications to the core Linux components are not supported. As a result, we can not directly modify the kernel and implement our DVFS control module into the Linux scheduler. Alternatively, we can achieve our goal by using the XenServer Driver Development Kit (DDK) to design and compile our DVFS controller. DDK is a VM that includes the tools for XenServer users to develop applications or drivers and generate the .iso images and package them into RPM packages. The generated packages can then be installed onto the XenServer host in Dom0 to perform the user defined tasks. We build our DVFS controller including the MAPI prediction model using the DDK. We use *gretl* [14], which is an open source time series analysis package written in the C programming language to help us build the ARIMA component in the prediction model.

5.4.1.1 Data Collection from CPU

At the local server level, performance counters [40], [41], [31] are used to capture the program behavior metrics in the proposed model Performance counters and counter control
registers are implemented as model specific registers (MSR). They can be accessed via the RDMSR and WRMSR instruction. For example, if we want to change the P-state of an Intel processor, where \( P\text{-state} \) represents a frequency and voltage operating state, we can use the WRMSR instruction to write a corresponding value of \( P\text{-state} \) to the \( IA32\_PERF\_CTL \) register, which is one of the Model Specific Registers (MSRs) [35], [36]. After writing the \( P\text{-state} \), the CPU will be unavailable for a short period of time due to voltage transitions. The CPU starts operating on the new frequency when the transition completes.

The CPU we use in our implementation is based on Intel Ivy Bridge architecture which contains seven performance counters per core [35], [36]. Performance counter one to four (PC1 to PC4) are fully programmable. These counters can count 116 different types of events respectively. The other three counters can each count one fixed type of event (for counter 5: IN-STR.RETIRED.ANY, 6: CPU_CLK_UNHALTED.CORE, and 7: CPU_CLK_UNHALTED.REF). In the implementation, performance counter one (PC1) records the number of memory accesses. PM2 records the number of stalled cycles. PM3 and PM4 record the number of instructions executed and the number of unhalted cycles respectively. Parameters in the model are then calculated based on the PC values, e.g., \( \text{MAPI} = \frac{\text{PC1}}{\text{PC3}} \), \( \text{CPI}_\text{exe} = \frac{(\text{PC4} - \text{PC2})}{\text{PC3}} \).

### 5.4.1.2 Optimal Frequency and MAPI Map

We can use the CPU frequency and MAPI model to calculate the optimal frequency for specific MAPIs given a SLA requirement. In this evaluation, we set the allowed performance loss compared to servers operating on highest frequency to 10\%, i.e., \( \text{target performance loss} \delta \) to 10\%. We first acquire the parameters we need for generating the map. LMbench [70] is used to measure the main memory latency and the time overhead of the DVFS operation in our system. Measurement results show the main memory latency is 80ns and the time overhead for each DVFS operation ranges from 150 to 250 \( \mu \text{s} \). The memory latency overlap \( \alpha \) is set to 0.9 in our system configuration. This is because for the CPU we use in our experiment (Intel Core i7 3770), the instruction issue rate is 6 with a 96 entry ROB. It takes 16 cycles for the reorder
Table 5.3: Optimal Frequency and MAPI MAP, $\delta$ to 10%, Intel Core i7 3770

<table>
<thead>
<tr>
<th>MAPI</th>
<th>Frequency (GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.007</td>
<td>3.7 GHz</td>
</tr>
<tr>
<td>(0.007-0.01]</td>
<td>3.4 GHz</td>
</tr>
<tr>
<td>(0.01-0.02]</td>
<td>3.3 GHz</td>
</tr>
<tr>
<td>(0.02-0.05]</td>
<td>3.1 GHz</td>
</tr>
<tr>
<td>(0.05-0.13]</td>
<td>2.8 GHz</td>
</tr>
<tr>
<td>(0.13-0.17]</td>
<td>2.4 GHz</td>
</tr>
<tr>
<td>(0.17-0.23]</td>
<td>2.1 GHz</td>
</tr>
<tr>
<td>(0.23-0.3]</td>
<td>1.9 GHz</td>
</tr>
<tr>
<td>0.3</td>
<td>1.6 GHz</td>
</tr>
</tbody>
</table>

buffer to be full and stall the CPU pipeline. In our system configuration, the memory latency is 160 cycles and when a memory access happens, the CPU can keep executing for 16 cycles before the ROB gets full. Thus $16/160=10\%$ of the memory latency is actually hidden by the out of order execution and $\alpha$ is set to 1-0.1=0.9. Although Intel did not publicize the P-states for i7 processors, we were able to identify 9 stable frequency and voltage pairs through writing to and observing the value of the $IA32_PERF_CTL$ register. Table 5.3 provides the mapping between optimal frequency and MAPI with 10% allowed performance loss.

5.4.2 CoolCloud Software Design

CoolCloud is mainly implemented in Java with 4500 lines of code which is publicly available on Github. The dynamic placement algorithm is implemented in 1700 lines of Java code. The VM placement controller is implemented using XenServer 6.5 SDK suite with 2300 lines of Java code. The VM controller software design includes three major components and communicates with XenCenter to manage the testbed data center. The first component is responsible of collecting runtime resource utilizations of each VM. These resources include CPU, memory, network and storage. This process is distributed to each VM where statistics are collected and sent to the data collector. The data collector then sends the VM statistics to the second component which is the heuristic algorithm mentioned in Section 5.3 that provides the
optimal VM placement solution. The third component is the VM migration commander which is responsible of sending out VM migration commands based on the placement solution from the placement algorithm. This component communicates with XenCenter and places each VM onto the optimal server. The DVFS controller is implemented in around 500 lines of C code excluding the ARIMA library. It is compiled through the XenServer DDK 6.5 and installed on each of the XenServers.

5.5 Experiment Result

We evaluate the proposed CoolCloud framework by running different workloads and observe the performance and energy savings. In order to thoroughly examine whether the DVFS controller and dynamic VM placement algorithm could effectively result in a energy efficient data center and maintaining SLA at the same time, long-running and fluctuating workloads are required to trigger the VM migration. We select benchmark programs from HiBench [85]. HiBench is a widely-used benchmark suite for Hadoop provided by Intel to characterize the performance of MapReduce based data analysis running in data centers.

We configure a Hadoop cluster built with Apache Hadoop 2.6.0 on top of our testbed data center to perform execution of MapReduce benchmarks. The cluster is composed of one master node and 20 computing nodes running Ubuntu 14.04. Each node has 4 GB of memory and 40 GB hard disk. The Hadoop configuration uses the default settings and runs with Oracle JDK 1.7. The estimation of resource usage of a specific VM is based on the VM’s history resource usage as all applications have program phases [63] that last for a period of time. With the characterization of the workloads and benchmark suite used in this paper, the workload fluctuations range from seconds to minutes, which are actively monitored by the CoolCloud data center. In this design, we use one minute as the threshold for a stable program phase and the threshold for initiating VM migration/remapping.
While the benchmark programs are running, the DVFS controller will keep record of the performance counters and calculate the MAPI values to adjust CPU P-states accordingly. The dynamic VM placement software will keep monitoring the VMs and servers to make migration decisions when necessary. At the same time, we keep record of each physical server’s resource utilization and power consumption for a one hour period. We run all experiments three times and use the average as the result.

Besides testing CoolCloud in a Hadoop environment, we also show how CoolCloud performs when it comes to managing VMs hosting Docker containers. Docker is a lightweight container that helps software developers to build, ship and run distributed applications. It uses resource isolation features of the Linux kernel such as cgroups and kernel namespaces to allow independent containers to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines. Docker containers usually run in VMs and this technology is becoming very popular in regard of fast development and deployment of cloud applications.

The power consumption of each physical server is measured based on the work in [17, 62], where the full-system average power is approximately linear with respect to CPU utilization as given in eq. (5.14).

\[ P_{Total} = P_{Dynamic} \cdot U_{Avg} + P_{Idle} \]  

(5.14)

It has proven to be an accurate way of measuring server power consumption especially in a data center environment where the total power consumption is an aggregation over a large number of servers. In our configuration, DVFS controller may adjust CPU frequency every second, thus the parameters of the energy model at the cluster level will be updated every time the P-state is modified. Table 5.4 provides the frequency steppings and corresponding voltage and power consumptions.

In the following experiment result charts, note that CoolCloud represents the proposed optimization design, 3.7GHz represents that all the CPUs in the data center will always run at 3.7GHz which provides a baseline for comparison. DVFS_only is when the data center is only provided with the DVFS controller design. Migration_only is when the data center
Table 5.4: Intel Core i7 3770 Frequency Steppings, Voltage and Power

<table>
<thead>
<tr>
<th>Frequency (GHz)</th>
<th>Voltage (V)</th>
<th>Power (w)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7 GHz</td>
<td>1.216 V</td>
<td>92</td>
</tr>
<tr>
<td>3.4 GHz</td>
<td>1.185 V</td>
<td>77</td>
</tr>
<tr>
<td>3.3 GHz</td>
<td>1.174 V</td>
<td>73</td>
</tr>
<tr>
<td>3.1 GHz</td>
<td>1.128 V</td>
<td>64</td>
</tr>
<tr>
<td>2.8 GHz</td>
<td>1.052 V</td>
<td>53</td>
</tr>
<tr>
<td>2.4 GHz</td>
<td>1.016 V</td>
<td>44</td>
</tr>
<tr>
<td>2.1 GHz</td>
<td>0.904 V</td>
<td>36</td>
</tr>
<tr>
<td>1.9 GHz</td>
<td>0.832 V</td>
<td>31</td>
</tr>
<tr>
<td>1.6 GHz</td>
<td>0.792 V</td>
<td>26</td>
</tr>
</tbody>
</table>

is only managed by the dynamic placement algorithm. This serves the purpose of measuring proportions of performance and energy savings provided by each individual component of the CoolCloud design.

Figure 5.7 provides the performance evaluation of CoolCloud. In this experiment, we measure the execution time for four benchmark programs from HiBench, i.e., WordCount, Sort, PageRank and Kmeans. For WordCount, the execution time is about the same across all three configurations, i.e., 115s for 3.7 GHz, 120s for DVFS only, 117s for Migration only and 122s for CoolCloud. CoolCloud requires slightly longer time to complete execution due to the overhead of live migration. However for PageRank, Kmeans, and TeraSort, CoolCloud demonstrates significant lower execution time compared to 3.7 GHz. This is because 3.7 GHz can not resolve the resource contention issue experienced by VMs. On the other hand, CoolCloud is able to detect the resource contention proactively and respond quickly by initiating VM migration to resolve this issue. Note that the cost for live migration is fully considered. The time duration for live migration typically ranges from several seconds to tens of seconds depending on the memory footprint of the VM. The small performance degradation comes from the live migration overhead and affects the applications performance running on that specific VM. CoolCloud prioritizes VMs’ with smaller memory footprints for migration thus significantly reduces the overall migration overhead.
Figure 5.7: CoolCloud Performance Evaluation

Figure 5.8: Energy Consumption Comparison
Another important thing to note in Figure. 5.7, is that the DVFS controller seems to be able to maintain the SLA agreement which is predefined as 10% performance loss. For WordCount, there is a 4.5% performance loss, 6.4% for PageRank and 7.3% for Kmeans. The only violation happens on TeraSort where the performance loss is 10.6%. This is mainly due to prediction errors from the ARIMA model. This problem can be resolved by adding feedback control into the DVFS controller, however due to the development limitation of XenServer, we were not able modify the kernel and keep track of each thread’s performance record.

Figure. 5.8 shows the energy consumption of the data center managed by the four different configurations. Each case is monitored in a 60 minutes time period. The data center starts with the same workload and initial VM placement. For WordCount, 3.7GHz consumes 42KJ of energy and CoolCloud consumes 39.6KJ which is a 5.8% energy saving. For TeraSort, 3.7GHz consumes 329KJ of energy while CoolCloud only consumes 233KJ. CoolCloud is able to achieve about 29.3% of energy savings. Also note that both DVFS\textsubscript{only} Migration\textsubscript{only} are able to provide energy savings and contributes to the overall energy savings of CoolCloud. The placement algorithm design constantly monitors the server resource utilization in a pro-active fashion, thus responding quickly to the workload fluctuations and seizing every energy saving opportunities. The reason that the total energy saving is not simply adding adding DVFS\textsubscript{only} and Migration\textsubscript{only} is because when these two designs work together, the P-state changes frequently which modifies the power consumption of VMs and servers. On the other hand, the VM migration will cause the program phases to change on a server which modifies the MAPI that the DVFS controller monitors. Note that the power consumption measured here is the result of taking all costs including the cost of live migration into consideration.

Figure. 5.9 and Figure. 5.10 show the evaluation of CoolCloud managing VMs hosting Docker containers. For this configuration, there are 20 VMs in total and we deploy 5 Docker apache web servers in each VM. The goal of this experiment is to observe how CoolCloud performs when different number of requests are randomly sent to the Docker servers. We prepare 3 web server scripts that will send requests to each docker servers. The number of
requests are randomly picked from 1K, 5K and 10K of requests. For example, Script 1 may send 10K requests to Docker server 5, 1K requests to Docker server 11, etc. Similarly, Script 2 may send 5K requests to Docker 5, 5K requests to Docker 11, etc.

Figure 5.9 provides the execution time of the 3 scripts under different management strategies. 3.7GHz takes the longest time to complete in all three Scripts. In Script 2, 3.7GHz takes 3652s to finish. We discovered the reason is that in Script 2, a large number of requests (140K) happened to be sent to the same server, resulting in high resource contentions among VMs residing on this server. This causes significant performance degradation when no migration strategies are used. However, CoolCloud is capable to quickly identify this SLA violation and moves VMs to servers with sufficient resource. On average, CoolCloud improves data center performance by 13.7%.

Figure 5.10 gives the energy consumption of three configurations running three scripts. 3.7GHz consumes the most energy due to resource contention among VMs. For Script 2, 3.7GHz consumes 893KJ of energy due to one server taking significant longer time to finish the requests. Again, CoolCloud consumes 666KJ of energy executing Script 2. It responds quickly to resource contentions and identifies VMs that have completed their request tasks and consolidates these VMs to a smaller number of servers to save energy. On average, CoolCloud achieves 19.0% energy savings compared to no energy design implemented.

### 5.5.1 Design Overhead and Improvement Breakdown

The design overhead of CoolCloud is mainly from the DVFS controller and the dynamic placement algorithm. As we are mindful of the overhead and the importance of computation efficiency in data center environments, we aim to minimize all the computations in each component. For the DVFS controller, the MAPI value is computed every second which requires to execute the ARIMA time series analysis. As we set the training sample to 50, the computation time for prediction is only about 3ms to 7ms which is only 0.3% to 0.7% of the one second execution time frame. With this minimum sacrifice, CoolCloud provides a significant
Figure 5.9: CoolCloud Performance Evaluation

Figure 5.10: Energy Consumption Comparison
improvement that it is mindful of the SLA requirement which is important for cloud service providers for maintaining quality of service.

For the dynamic placement algorithm design, the computation time for finding optimal placement solutions depend on the number of VMs. We observed that in the case of 20 VMs, it takes 180ms to calculate the placement decisions. When there are 50 VMs, the computation time is about 630ms. Based on our simulation study, as the number of VMs increase to 1000 VMs, it takes the algorithm 22s to find the placement solution. Overall, the dynamic placement design is a practical design that can find VM mapping solutions efficiently in a data center environment.

5.6 Related Work

5.6.1 DVFS Designs

A number of works have used DVFS related techniques to provide energy efficient computing, we limit our discussion to the methods that are most relevant to our work. Recent research on DVFS based energy efficient techniques can be classified into at least three groups. The first group of techniques use known task arrival times, workload, and deadlines to implement algorithms at the task level or operating system [37, 56, 68, 30]. Horvath et al. [30] proposed a DVFS policy for multi-tier web server system that can minimize global energy consumption while meeting the multi-stage end-to-end delay constraint. Isci et al. [37] analyzed different policies for chip level power management under a specific power budget. These policies adjust power modes of individual cores targeting at different objectives such as prioritization of cores/benchmarks, balancing power among cores and optimizing system throughput.

The second group of techniques use compiler or application support for performing DVFS [59, 22, 52, 55]. For example, in [92], the authors provide an application level power management by using the knowledge provided by the application to save energy. In [59], the authors use dynamic profiling of branch probability to characterize workload then use DVFS to main-
tain power-performance balance. This group of methods need additional code added to the application before it is executed on the system.

The last but not the least group of techniques use program runtime characteristics or statistics to identify the workload of a task. Then estimate and predict the optimal voltage and frequency setting [50, 11, 10, 47, 5]. For example, Kotla et al. [50] use the program runtime information instruction per cycle to decide the running frequency, this method can reduce energy waste caused by memory stalls, however the scheme does not guarantee the SLA requirement. These techniques can be further classified as fine-grained or course-grained. Course-grained techniques determine the voltage and frequency setting on a task-by-task basis. Fine-grained techniques adjust the voltage and frequency setting within a task boundary and usually perform better than course-grained techniques.

5.6.2 Energy Aware VM Placement

Virtualized data center management has gathered a great amount of research interests in the past few years. Recent studies focus on improving server resource utilizations, meeting power budgets, balancing workloads among servers and reducing any energy related costs. We have done an extensive study of past works to inspire our design. A brief discussion of past achievements and limitations is given as follows.

Grit et al. [24] consider some VMs replacement issues for resource management policies in the context of a system for on-demand leasing of shared networked resources in server clusters. When a migration is not directly feasible, due to sequence issues, the VM is suspended using suspend-to-disk. Once the destination server becomes available, the VM resumes. In our work, when migration is not feasible, we first try to find the delay-tolerant VM and suspend it to release server resource for other time-sensitive VMs. When server resource becomes available, the suspended VM resumes. Electricity price is also considered for determining when to resume the suspended VM in order to minimize cost. Luo et al. [58] presents a two-stage design and the eco-IDC (Energy Cost Optimization-IDC) algorithm to exploit the
temporal diversity of electricity price and dynamically schedule workload to execute on IDC servers through an input queue. Wu et al. [86] presents a scheduling algorithm with dynamic voltage frequency scaling technique to increase resource utilization in data centers.

Jing et al. [88] propose a multi-objective virtual machine placement algorithm that simultaneously minimize power consumption, resource wastage and thermal dissipation. Xin et al. [53] also consider physical resources as multi-dimensional and propose a multi-dimensional space partition model to determine the mapping of VMs and PMs. [77, 64, 9, 88, 53] are static VM placement methods that only consider the initial VM placement and do not consider future dynamic workload changes that may need VM remappings. Shrivastava et al. [73] consider the inherent dependencies between VMs comprising a multi-tier application. They propose a scheme called AppAware to determine VM placement that can greatly reduce network traffic considering the interaction between applications running on different VMs.

Meng et al. [63] consider the network traffic and bandwidth as factors that may affect system performance. They optimize VM placement based on traffic patterns and communication distances. VMs with mutual bandwidth usage are assigned to PMs. Cost-aware workload placement is also gathering wide interest in data center operations. [6, 61, 82, 51, 57] propose to reduce data center operational cost by exploiting electricity price differences across regions. Workload can be migrated to a data center where resource is sufficient and energy price is low. However, migration cost and service level agreement are not considered. Furthermore, short running workloads do not make sense in this scenario. [7, 67, 34] focus on how to use green energy to power data centers.

5.7 Conclusion

This paper presents an energy aware design for data center environments. The proposed CoolCloud framework optimizes energy consumption from two levels: the cluster level and the local server level. At the cluster level, a computation efficient algorithm is proposed to find
optimal VM placement solutions according to VM resource requirement. At the local server level, a DVFS controller is proposed to adjust CPU voltage and frequency according to program phases. The CoolCloud provides energy saving while maintaining SLA requirements. A testbed data center that represents current enterprise data center infrastructure is build to evaluate the design. Evaluation against Hadoop benchmark programs and Docker applications shows that the design can effectively provide a energy efficient data center for today’s cloud computing services.
CHAPTER 6. CONCLUDING REMARKS AND FUTURE WORK

In the preceding chapters, we present our research achievements on: 1. how to achieve energy savings in a virtualized data center environment; 2. how to maintain service level agreements; 3. how to make our design practical for actual implementation in enterprise data centers. Chapter 2 presents the feedback control based DVFS scheduler that can provide CPU energy savings exploiting application’s program phases while keeping the system performance under the service level agreement. Chapter 3 and 4 presents the study of using integer linear programming to find optimal VM placement solutions and the heuristic algorithm design for solving the placement problem computation efficiently. Chapter 5 combines the research of optimizing energy at the local server level and the cluster level and presents the optimization framework named CoolCloud to minimize energy consumption in virtualized data centers with the service level agreement taken into consideration. We summarize the main contributions and discuss proposed methods for the thesis work.

- We propose a cool scheduler that can be used in enterprise data centers to provide CPU energy saving under the specified SLA requirement by exploiting the applications’ run-time program phases.

- The proposed scheduler greatly improves the computation efficiency compared to two of the most advanced related works and significantly reduces the number of unnecessary DVFS operations which are ignored in recent works.
• The proposed scheduler is based on feedback control which can precisely control the performance loss and maximize energy saving with the SLA requirement always guaranteed.

• We propose a dynamic virtual machine framework with the objective to minimize energy consumption. The proposed framework actively monitors workload runtime fluctuations, and provides dynamic placement solutions.

• The dynamic VM placement design considers server resource utilization in all aspects that eliminates energy wastage and performance bottlenecks caused by resource wastage.

• We provide a heuristic design based on simple quick sort and greedy algorithm that achieves near-optimal energy savings with low computation complexity. This makes our design a practical solution for large size data centers.

• We conduct a comparison with industry leading design VMware’s DRS to demonstrate the effectiveness of our design in regard of performance and energy savings.

• We provide a prototype implementation of the CoolCloud data center using the most advanced tool stacks available. CoolCloud integrates dynamic virtual machine placement and DVFS control.

• We choose workloads from web service applications, big data benchmarks, i.e., HiBench to Docker software containers that represent today’s cloud computing environment to thoroughly evaluate CoolCloud.

Based on the research presented in this thesis, we have many other future research opportunities: 1. Include I/O phases as a guidance for adjusting DVFS at the local server level. Similar to using MAPI, we can define I/O intensive phase and computation intensive phase and propose a model to identify the desired CPU frequency according to the program phase. 2. The DVFS designs presented in this dissertation can be applied to end users’ Android mobile devices for
energy savings. Recent advances in ARM processors also support per core DVFS, thus making our DVFS designs easily portable to ARM based processors. Compared to a desktop or server environment, Android usually has much less numbers of processes executing, which makes it much easier to maintain application SLAs. 3. Include storage migration in the dynamic VM placement framework to enhance the robustness and flexibility of virtualized data centers. 4. Add a coordination design between cluster level and local server level optimizations for more fine grained SLA control. 5. Consider the electricity price fluctuation in our optimization framework, which will lead to the most cost efficient way of operating the data center.
Bibliography


[40] Isci, C., Contreras, G., and Martonosi, M. (2006c). Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In the


on Information communications, INFOCOM’10, pages 1154–1162, Piscataway, NJ, USA.
IEEE Press.


