Performance-aware Task Scheduling in Multi-core Computers

by

Shenyuan Ren

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

PhD

Department of Computer Science

The University of Warwick

July 2018

Abstract

Multi-core systems become more and more popular as they can satisfy the increasing computation capacity requirements of complex applications. Task scheduling strategy plays a key role in this vision and ensures that the task processing is both Quality-of-Service (QoS, in this thesis, refers to deadline) satisfied and energy-efficient. In this thesis, we develop task scheduling strategies for multi-core computing systems. We start by looking at two objectives of a multi-core computing system. The first objective aims at ensuring all tasks can satisfy their time constraints (i.e. deadline), while the second strives to minimize the overall energy consumption of the platform. We develop three power-aware scheduling strategies in virtualized systems managed by Xen. Comparing with the original scheduling strategy in Xen, these scheduling algorithms are able to reduce energy consumption without reducing the performance for the jobs. Then, we find that modelling the makespan of a task (before execution) accurately is very important for making scheduling decisions. Our studies show that the discrepancy between the assumption of (commonly used) sequential execution and the reality of time sharing execution may lead to inaccurate calculation of the task makespan. Thus, we investigate the impact of the time sharing execution on the task makespan, and propose the method to model and determine the makespan with the time-sharing execution. Thereafter, we extend our work to a more complex scenario: scheduling DAG applications for time sharing systems. Based on our time-sharing makespan model, we further develop the scheduling strategies for DAG jobs in time-sharing execution, which achieves more effective at task execution. Finally, as the resource interference also makes a big difference to the performance of co-running tasks in multi-core computers (which may further influence the scheduling decision making), we investigate the influential factors that impact on the performance when the tasks are co-running on a multicore computer and propose the machine learning-based prediction frameworks to predict the performance of the co-running tasks. The experimental results show that the techniques proposed in this thesis is effective.

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor Dr. Ligang He, whose guidance, encouragement and support have been invaluable to me during my time at the Department of Computer Science at the University of Warwick. I benefited greatly from his insightful advices and comments in finding and solving research problems. I look forward to maintaining our collaboration in the future.

I would like to thank my husband Junyu whose patience, encouragement and unwavering love has been the magical special ingredient in my life for the past two years. I thank my parents for their continuous and unreserved support in my pursuit of an academic career.

I thank Dr. Roger Packwood for his support in solving technical issues and his positive attitude which influenced me greatly. I thank the staffs in our department (Ruth Cooper, Sharon Howard, Jackie Pinks, Jane Clarke, Mike Cribdon, Maria Ferreiro, Lynn McLean, Gillian Reeves-Brown and Paul Williamson) for their kind help and warm supports.

Last but not the least, I want to thank my fellow lab-mates, particularly Bo Gao, Chao Chen, Huanzhou Zhu, Zhuoer Gu, David Purser, James Van Hinsbergh, Peng Jiang, Chen Gu, Bo Wang, Qingzhi Ma, Yijun Quan, Haoyi Wang, Hao Wu, Zhiyan Chen and Yujue Zhou for their stimulating discussions in current trends in technology and for creating all the happy memories that we share.

Declarations

This thesis is submitted to the University of Warwick in support of the authors application for the degree of Doctor of Philosophy. It has been composed by the author and has not been submitted in any previous application for any degree. The work presented was carried out by the author except where acknowledged. Parts of this thesis have been previously published (or accepted) by the author in the following:

1 S. Ren, L. He, H. Zhu, Z. Gu, W. Song and J. Shang. Developing power-aware scheduling mechanisms for computing systems virtualized by Xen. *Concurrency and Computation: Practice and Experience*, 29(3): e3888. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3888.

2 S. Ren, L. He, J. Li, C. Chen, Z. Gu, and Z. Chen. Scheduling dag applications for time sharing systems. *18th International Conference on Algorithms and Architectures for Parallel Processing.*

3 S. Ren, L. He, J. Li, Z. Chen, P. Jiang, and L. Chang-Tsun. Contention-aware prediction for performance impact of task co-running in multicore computers. *Wireless Networks Journal.*

4 P. Jiang, L. He, S. Ren, and Z. Chen. vchecker: a application-level demand-based co-scheduler for improving the performance of parallel jobs in xen. *9th EAI International Conference on Big Data Technologies and Applications.*

5 J. Li, L. He, and S. Ren. Data fine-pruning: A simple way to accelerate neural network training. *15th IFIP International Conference on Network and Parallel Computing.*

6 P. Jiang, L. He, S. Ren, Z. Chen, and R. Mao. vplacer: a co-scheduler

for optimizing the performance of parallel jobs in xen. *18th International Conference on Algorithms and Architectures for Parallel Processing.*

Sponsorship and Grants

The research presented in this thesis was made possible by the support of the following benefactors and sources:

- China Scholarship Council (CSC)

Abbreviations

AWS	Amazon Web Service
BFM	Best Frequency Match
BFMM	Best Frequency Match for Multi-core
BFMS	Best Frequency Match for Single-core
BT	Block Tri-diagonal solver
BVT	Borrowed Virtual Time
BoT	Bag-of-Tasks
CG	Conjugate Gradient
CMPs	Chip Multi-core Processors
CP	Critical Path
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
ddl	Deadline
DER	Desired Execution Requirement
DNA	Deoxyribonucleic Acid
DVFS	Dynamic Voltage Frequency Scaling
EDF	Earliest Deadline First
EP	Embarrassingly Parallel
GA	Genetic Algorithm
IaaS	Infrastructure as a Service
IBM	International Business Machines
IDC	International Data Corporation

IoP	Internet of People
IPAC	Infrared Processing and Analysis Center
LLC	Last Level Cache
LLS	Least Performance Loss
LSVMs	Latency-sensitive Virtual Machines
LTF	Largest Task First
LU	Lower-Upper Gauss-Seidel solver
NASA	National Aeronautics and Space Administration
NLS	No performance Loss Scheduling
NP	Non-deterministic Polynomial-time
NPB	NASA Advanced Supercomputing Parallel Benchmarks
OS	Operating System
PE	Performance Event
PM	Physical Machine
P-states	Power-states
PSO	Particle Swarm Optimization
QoS	Quality-of-Service
RL	RunningList
RNA	Ribonucleic Acid
RT-Xen	Real Time Xen
SEDF	Simple Earliest Deadline First
SPEC	Standard Performance Evaluation Corporation
SVM	Support Vector Machine
TAA	Task Allocation Algorithm
TCP	Critical Path Execution Time

TLB	Translation Lookaside Buffer
TmpRT	Temporary Remaining Time
TS	Time-sharing
VCPU	Virtual Central Processing Unit
VM	Virtual Machine
VMM	Virtual Machine Monitor
WCET	Worst-case Execution Time

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Virtual Machine Scheduling in Cloud Computing Platforms

Cloud computing emerges as one of the most important technologies for interconnecting people and building the so-called "Internet of People (IoP). Nowadays, energy consumption in such a system is a critical metric to measure its sustainability and eco-friendliness. Indeed, data centers used by the Cloud service providers have become one of the fastest growing sources of power consumption in industry. According to IDC (International Data Corporation), power consumption of data centers worldwide accounts for about 8% of the global electricity, which does not include the additional electricity consumed by the cooling systems equipped in the data center. In such a Cloud-based IoP, the virtualization technique, which allows multiple operating system instances (i.e., Virtual Machines) to run simultaneously in a physical machine, provides the key supporting environments for running the IoP jobs such as performing data analysis and mining personal information.

Xen [68] is a popular virtualization hypervisor used in the academic community. It has also been widely deployed in a number of industry-level Clouds, such as AWS (Amazon Web Service), Rackspace, Verizon, etc. SEDF (Simple Earliest Deadline First) is a scheduler in Xen. In SEDF, the CPU requirement for each VM is specified by a tuple (s,p,x), in which s and p designate that the VM has to run at least s in a period of p. This CPU requirement can be translated to the deadlines by which a VM has to start running (otherwise, the CPU requirement will not be met). In each scheduling round, SEDF puts the

VM with the earliest deadline into execution [15].

Dynamic Voltage Frequency Scaling (DVFS) is a power management technique. DVFS can change the running frequency of CPU dynamically as required and therefore reduce power consumption when the tasks do not need to be run with the maximum CPU frequency. Xen currently has four power governors. 1) The Ondemand governor selects the CPU frequency which best fits the VM (guest domain). 2) The Userspace governor selects the frequency specified by user. 3) The Performance governor selects the highest frequency. 4) The Powersave governor selects the lowest frequency. There are twelve frequency states in the Xen power management, in which the state P0 represents the highest frequency while the state P12 represents the lowest frequency. Here, we consider the most complex scenario and assume that the guest domains are run under the Ondemand governor, namely, the CPU frequency is dynamically adjusted towards the best execution frequency of a guest domain.

In Xen, the CPU frequency can only be changed by one state in every interval of 10ms. The interval of 10ms is called the *frequency scaling slice*. For example, it takes Xen 40ms to change the CPU frequency from P1 to P4. Our studies show that this limitation in Xen in frequency changing may cause the energy waste and performance loss (The problem is illustrated by an example the motivation section of Chapter 3), which this work aims to reduce. In this work, we conduct the theoretical ananlysis and construct the performance model and the energy consumption model by taking into account the feature of Xen in frequency changing. Based on the analysis and the models, we derive the condition under which the best performance can be achieved, i.e., there is no performance loss caused by the limitation of Xen in frequency changing. Further, we propose a frequency-aware scheduling policy, called BFM (Best Frequency Match), by adapting the SEDF scheduling policy in Xen. Compared with SEDF, BFM is able to reduce the power consumption of running VMs without violating their CPU requirements.

1.2 Scheduling DAG Applications for Time Sharing Systems

DAG is often used to model the precedence constraints of a group of related tasks. Many DAG (Directed Acyclic Graph) scheduling algorithms have been proposed in literature. The makespan of a DAG is an important metric to measure the performance of a DAG scheduling solution. When computing the makespan of a DAG, it is typically assumed that the tasks scheduled on the same computing node run in sequence, i.e., being executed one by one in the computing node (which we call the *sequential execution* in this thesis) [48][50][93]. This assumption is reasonable in the cluster platform, where there is only a central queue in the head node and a new task is sent to a computing node only when the node has completed the execution of the existing task. However, in some situations, such as distributed systems and virtualized environments, there may not be a central queue in the system. In a distributed system, there is no a centralized management mechanism. The tasks in a DAG are often sent to the computing machine as designated in the scheduling solution. After the computing machine receives these tasks, the tasks are run in the time sharing manner by the operating system. In virtualized environments, a VM is often created to run a task. When multiple tasks are scheduled to the same machine, there will be multiple VMs co-running in the physical machine. These VMs will not be executed in sequence, but concurrently (i.e., time sharing) by the schedulers (such as Credit or SEDF) deployed in the Virtual Machine Monitor.

Our studies, the details of which are presented in Chapter 4, show that the discrepancy between the assumption of sequential execution and the reality of time sharing execution may lead to inaccurate calculation for the finish times of individual tasks and further for the execution performance, such as in terms of makespan, of the whole DAG.

In this work, we first investigate the key difference between the time-sharing execution and the sequential execution, and reveal the impact of the time sharing execution on the DAG makespan. Based on the analysis, we adapt the conventional method of computing the DAG makespan in the sequential execution and present our counterpart makespan model and method in the time-sharing execution. Usually, the makespan in the time sharing execution is worse (longer) than that assumed in the sequential execution. Therefore, we propose the new DAG scheduling strategies (a task migration algorithm and a task allocation algorithm) for time-sharing systems.

1.3 Performance Impact of Task Co-running in Multicore Computers

In the task scheduling, it is often assumed that the scheduler knows the execution time of the tasks. However, it is a non-trivial task to product the accurate performance prediction for tasks, although a number of techniques are indeed developed to predict the task performance [57]. These three situations make it even more challenging to predict the task performance: 1) task co-running: the tasks are running simultaneously (co-running) on multiple CPU cores in a multicore processor; 2) scaled CPU frequency: the CPU frequency may be scaled to run tasks; 3) time-sharing execution: the tasks are running in a time-sharing manner on the same core, due to the following reasons.

There are resource contention and interference among the co-running tasks, since they need to share (contend) the resources in the computer such as internal buses, cache, memory, hard disk, etc. The resource contention may lead to the longer completion times of the tasks. The resource contention relation is complicated because both the intensity level of contention and the type of the contention (i.e., which type of the resource is contended by the tasks most intensively) do not only relate to the hardware specification of the system (such as cache size and memory bandwidth), but also vary from the characteristics of the co-running tasks (such as memory access frequency, I/O requirement and cache usage). Different co-running combinations of tasks may lead to very different contention levels of intensity. The complexity nature of the contention among co-running tasks makes it difficult to develop the static formulas for accurate performance prediction.

On one hand, executing with lower frequency can reduce the power consumption. On the other hand, the execution time of the task will increase. Many researches have been conducted based on the trade-off between the task execution time and the energy consumption. We conducted experiments to show that the relationship between CPU frequency and execution time is non-linear. Based on our investigation, we found that the performance impact of running with different CPU frequencies depends on the characteristics of the task, as well as the architecture of the hardware.

A time sharing scheduler aims to provide all processes with relatively equal interval of time to access the CPU. It allows more efficient use of the computer hardware; where a program is waiting for some events such as a user input or I/O operation to complete, the central processor can still be used to run another program [95]. Time-sharing execution often achieves higher CPU utilization than sequential execution. However, due to the resource contention and other resource requirements during processing, different combinations of time-sharing executions may lead to different performance (This problem is illustrated by an example in motivation section of Chapter 5).

We investigate the performance impact of tasks in the scenarios mentioned above and present the method to identify the influential factors for the given co-running tasks. Further, we propose a machine learning-based approach to predicting the performance of the tasks. Two prediction frameworks are developed for two types of task that are often seen in production systems: repetitive tasks (i.e., the tasks that arrive at the system repetitively) and new tasks (i.e., the task that are submitted to the system the first time), the difference between which is that we have the historical running information of the repetitive tasks while we do not have the prior knowledge about new tasks.

Given the limited information of the new tasks, a two-stage online prediction framework is developed to predict the performance of the tasks by sampling the performance events on the fly for a short period and then feeding the sampled results to the prediction framework. We conducted the extensive experiments with the SPEC2006 benchmark suite to compare the effectiveness of different machine learning methods and present our observations and analyse. The results show that our prediction model can achieve the high accuracy for both repetitive tasks and new tasks in the three scenarios summarized above: tasks co-running, varied CPU frequency and time-sharing execution.

1.4 Thesis Organization

The reminder of this thesis is organized as follows. This chapter provides a brief overview of the work conducted in this thesis. In the next chapter, we present the literature review of our research topics. Thereafter, each of the three chapters (Chapter 3-5) of this thesis presents the task scheduling strategies from a different scenario.

In Chapter 3, we formally construct the models for execution time and power consumption. We identify the best performance scheduling situation where there is no performance loss. Given the execution order of the tasks, we can determine the start frequency that can guarantee that every task is able to run with the best performance frequency. Thereafter, we propose a power consumption-aware scheduling policy and design the task scheduling policies on a single core and also on multiple cores. Our scheduling policies are able to reduce power waste, which is supported by our intensive experiments.

In Chapter 4, firstly, we present a motivating case study to demonstrate the difference of the time sharing execution from the sequential execution and its impact on the makespan. Secondly, we present the workload and system model and the notations used in our scenario. Then we present the makespan models with both sequential and time-sharing executions, followed by the task adjusting algorithm and the DAG allocation algorithm for the time-sharing execution.

Experimental results are presented in the last section of this chapter.

In Chapter 5, we investigate the influential factors of performance impact in the three scenarios (i.e. task co-running, scaled CPU frequency and time-sharing execution); a performance prediction framework is developed for repetitive tasks and new tasks. Then we introduce the machine learning approaches applied in our framework. The next section shows the experimental results of our work. The experiments are conducted on benchmarks SPEC 2006 and NPB. Finally, we conclude this chapter.

Lastly, Chapter 6 presents the conclusions and talks about the future work.

CHAPTER 2
Literature Review

This chapter presents the literature reviews. Section 2.1 and Section 2.2 discuss the scheduling strategies for the virtualized environments and native environments, respectively. Then we reveal the problem of performance degradation when scheduling co-running tasks and present the related research work in Section 2.3.

2.1 Scheduling Strategies for Computing Systems Virtualized by Xen

Xen is a hypervisor developed by the University of Cambridge. It is widely used by many popular cloud service providers such as Amazon and IBM SoftLayer. The Xen Hypervisor supports several different virtual CPU schedulers: Borrowed Virtual Time (BVT) Scheduler, Simple Earliest Deadline First (SEDF) Scheduler, Credit Scheduler, Credit2 Scheduler, etc. SEDF uses a real-time scheduling algorithm called "Earliest Deadline First (EDF)" to guarantee the CPU requirements of the VM. EDF is an optimal scheduling algorithm in the following sense: if a set of independent tasks (each task has its corresponding arrival time, execution time and deadline) can be scheduled in a way that there is no deadline miss, the EDF can schedule this set of tasks.

When scheduling periodic processes without violating their deadlines, EDF has a utilization bound of 100%, which can be represented by the following expression:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2.1}$$

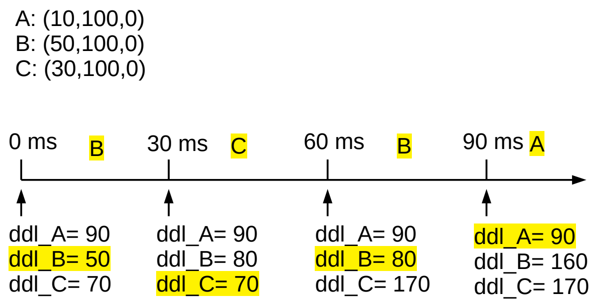


Figure 2.1: An example of SEDF scheduling process

where U is the CPU utility, $\{C_i\}$ are the worst case execution times of the n tasks and the T_i is their relative deadlines. That is, EDF can guarantee that all deadline constraints are satisfied and the total CPU utilization is not more than 100%.

In the SEDF scheduler, the CPU requirement for each VM is specified by a tuple (s,p,x), in which s and p that the VM has to run at least s units in a period of p units. This CPU requirement can be translated to the deadlines by which a VM has to start running (otherwise, the CPU requirement will not be met). In each scheduling round, SEDF puts the VM with the earliest deadline into execution [15]. Please note that the deadline in our first work (presented in Chapter 3) refers to a relative deadline, which represents the CPU requirement of the tasks.

Figure 2.1 illustrates an example of the SEDF scheduling process. We assume that task A, B, and C are scheduled by SEDF. The scheduling slice is 30 ms. The CPU requirements of the tasks are A(10,100,0), B(50,100,0) and C(30,100,0), respectively. Namely, task A, B and C need to run for at least 10, 50 and 30 ms in every 100 ms, respectively. At time point 0ms, the relative deadline of the tasks (denoted by ddl_x) are shown in the figure. At each scheduling point, SEDF will select the task with the earliest relative deadline to execute (task B in the figure). After running for 30ms, the relative deadline of task B changes to 80ms while others stay the same. At time point 30ms, SEDF will select task C to execute since task C has the earliest relative deadline. The same scheduling policies are applied at other scheduling points.

Credit scheduler is a proportional fair share scheduler for virtual CPUs. Each virtual machine is assigned a weight and a cap. The cap refers to the amount (expressed as percentage) of CPU that a VM can use while the credit is represented by the weight. The default time slice of the Credit scheduler is 30 ms. Once every 30 ms, the credits of all runnable VMs are recalculated, the executing VM will be appended to the end of the runnable queue and the scheduler will pick the first runnable VM to execute. Only the VMs with non-zero credits are runnable. Further, to improve the performance of I/O intensive tasks, Credit scheduler assigns them a priority boost. The VMs that are waken up are likely to be scheduled immediately.

There have been many previous efforts to optimize the performance and the power consumption of Xen-based systems. This section discusses several performance models and power consumption models, the energy-aware scheduling strategies, and some methods for optimizing the resource consolidation on multi-core processors.

2.1.1 Performance Model and Power Consumption Model

Reference [46] constructs an energy-aware stochastic task scheduling architecture in heterogeneous computing environments, which incorporates HCS, BoT applications, stochastic tasks, an energy model, and time and energy budget constraints into consideration. Reference [92] [62] [6] and [104] also present power consumption models in different virtualized environment.

2.1.2 Energy-aware Scheduling Strategy

RT-Xen [100], which is the first real-time hypervisor scheduling framework for Xen. It bridges the gap between real-time scheduling and Xen. Basing on Xen's widespread adoption, RT-Xen constructs an attractive platform for integrating an extensive range of real-time and embedded systems.

Reference [43] attempts to meet the execution requirements of all the tasks and to minimize the overall energy consumption of the processor. It considers the issue of scheduling a set of aperiodic tasks on DVFS-enabled multi-core processors. Giving an ideal case to define the tasks' Desired Execution Requirement(DER), the algorithm will points out an evenly allocation method to achieve saving processor energy, where it is especially suitable for real-time systems.

Reference [51] presents a task scheduling strategy that solves the task allocation problems on DVFS-enabled CPU cores, the tasks execution order and the CPU processing frequency for each task. It formulates the task scheduling model, the energy consumption model, a CPU processing frequency model and a cost function. The formulas deduces the scheduling policy for both batch mode and online mode of the tasks. The algorithm guarantees the minimal total cost for every time interval.

In[41], a power consumption model is purposed for estimate the energy consumption of the tasks in cloud systems. It divides the power consumption into leakage power and dynamic power and formulates the processor energy consumption according to each in-processor event. The work presents an energy-credit scheduler that schedules the tasks according to their energy budgets instead of time credits. Also their scheduling algorithm lacks the careful consideration of the tasks' executing time requirements.

Reference [91] provides the performance model, the compute cluster model and virtual machine model on DVFS-enabled cluster tasks. In addition, it presents a power-aware cluster scheduling algorithm to minimize the power consumption.

Reference [70] extends an existing formulation of the power-aware job placement problem as to account for DVFS-enabled cluster nodes. Reference [70] discusses two optimization problems: (i) optimizing performance by given a constraint on energy consumption; (ii) optimizing the energy consumption by given a constraint of job performance. In addition, it calculates the bound for several instantiations of the DVFS model, thereby quantifying the added benefit of increased DVFS capabilities.

2.1.3 Performance-aware Scheduling Strategy

Reference [105] presents an optimization technique: periodically coalescing and handling I/O events, which guarantees I/O performance as well as reduces the preemption rate and scheduling latency. It also uses a Round-Robin manner to handle I/O events periodically which improves I/O performance of computational tasks, though it leads to increasing number of migrations.

In [114], a layered graph is built to illustrate the co-scheduling problem. As a result, the problem of finding the optimal co-scheduling strategy is then modelled as looking up a shortest valid path in the built graph. Further, their work presents the approaches of seeking the shortest path for both serial jobs and parallel jobs. Besides, several optimization measures are also designed to speed up the solving process.

Reference [2] emphasizes the virtual time discontinuity problem for lock and interrupt handling in guest operating systems. It presents the downside of sub-millisecond time slicing, and the architectural implication for future support of VM consolidation and also proposes a context preservation technique based on time sampling.

To mitigate I/O processing latency while retaining the benefit of CPU sharing, reference [101] provides a new class of VMs named latency-sensitive VMs (LSVMs). It provides a better performance for I/O-bound applications while using the same resource share as other CPU-sharing VMs. LSVMs are powered by vSlicer, which is a hypervisor-level methodology scheduling each LSVM more frequently but with a shorter micro time slice, without breaking the CPU fairness among all sharing VMs.

Reference [27] presents a Flat Lightweight File System (iFlatLFS) to manage small files basing on a simple metadata scheme and a flat storage architecture. It can greatly simplify the original data access procedure. The proposed metadata holds as little as a fraction of the metadata size used on traditional file systems.

In [40], a task-aware virtual machine scheduling strategy is presented based on inference methods applying the gray-box knowledge. It introduces partial boosting that is a priority boosting mechanism with task-level granularity. By applying such technique, any I/O-bound tasks can be selectively scheduled to handle their incoming events straightway. As a result, the work using lightweight mechanisms with full CPU fairness among VMs enhances I/O-bound tasks performance within heterogeneous workloads.

vGreen [22] is a multi-tiered software system proposed for energy-efficient computing in virtualized environments. It presents a number of novel hierarchical metrics measuring power and performance characteristics of both physical and virtual machines. In addition, several policies are introduced for energy efficient virtual machine scheduling across the whole deployment.

Reference [35] addresses the challenges of dynamically scheduling parallel jobs with QoS demands (soft-deadlines) in multi-clusters and grids system. Three metrics (over-deadline, makespan and idle-time) are consolidated with appropriate weights to evaluate scheduling performance. Moreover, two levels of performance optimization methods are developed for the multicluster environment.

2.2 Scheduling Strategies for Tasks in Native Environment

In this section, we discuss the task scheduling strategies in native environments (as opposed to virtualized environments). The tasks can be broadly divided into two classes: Bag of Tasks, which refers to a set of independent tasks, and DAG, which represents a set of tasks with precedence constraints. In this subsection, we discuss scheduling strategies in native environments for these two classes of tasks.

2.2.1 Scheduling Strategies for Bag of Tasks (BoT) in Multi-core Computers

Several works have been conducted to schedule BoT applications to multi-core processors [60] [72] [79] [113] [80] and [16]. Reference [81] builds an energy-aware cloud computing platform including its architecture, job and power consumption model. Then, the authors develop a prediction approach to forecast the short-term workload that combines the linear regression and wavelet neural network techniques. Reference [71] develops a comprehensive multi-objective optimization model that takes four conflicting objectives, namely minimizing task transfer time, task execution cost, power consumption, and task queue length, into consideration. Reference [84] investigates BoT scheduling from an energy efficiency perspective.

Survey [4] presents the power management strategies that minimize the makespan under a power budget. Reference [107] addresses the energy consumption problem by developing DVFS-based scheduling strategies for parallel real-time tasks. Reference [52] develops task scheduling strategies that find a balance point between two conflicting objectives (performance and energy consumption). Reference [85] studies BoT load balance from an energy efficiency perspective. In addition, two execution modes (batch mode and online mode) are considered in the work. Reference [116] designs a dynamic voltage scaling-based scheduling strategy called adaptive energy-efficient scheduling (AEES), for BoT on heterogeneous clusters.

2.2.2 Scheduling Strategies for DAG Applications in Multi-core Computers

It is typical to run a DAG application on clusters in order to exploit the inherent parallelism in the DAG topology. Several popular scheduling frameworks have been developed on clusters: YARN [87], Borg [89], Sparrow [65], Apollo [7], Mercury [39], etc. The centralized scheduling frameworks such as YARN and

Borg only have the global queues. In order to improve the scheduling performance, it now becomes increasingly popular to employ the distributed scheduling in large-scale data centres, where multiple schedulers make the scheduling decisions for different types of jobs simultaneously and independently. Such distributed scheduling frameworks include Mercury [73], Apollo and our previous work presented. In distributed scheduling frameworks, a PM may receive the tasks dispatched by different schedulers and these tasks are typically run in a time sharing manner in the node. The experiments presented in [110] also indicate that the vast majority of the PMs in such clusters run multiple tasks concurrently.

Scheduling a DAG and minimizing its makespan are proven to be a NP-complete problem when there are more than two PMs [28]. Thus many heuristic and meta-heuristic scheduling approaches are developed to minimize the DAG makespan [13][47][31][9][37].

Although scientists began to study the scheduling long time ago, it is still a hot topic nowadays to investigate the scheduling strategies for new platforms and scenarios emerging over time, such as virtualized systems [99], multi-sites workflow scheduling [55], soft real-time scheduling in data centres [44], energy-aware scheduling [66], and the scheduling with multiple objectives on IaaS Clouds [66]. Reference [106] proposes a scheduling algorithm, which combined the cluster-based method and the interval insertion strategies to solve the problem that most of the researches ignore the allocation of the non-critical predecessors. Reference [76] addresses the problem of scheduling periodic parallel tasks with implicit deadlines on multi-core processors. Reference [86] proposes an aging-aware task scheduling framework for NoC-based multi-core systems. It develops a particle swarm optimization (PSO)-based heuristic to solve the scheduling problem with an optimization objective of total task completion time, and finally obtain a scheduling result with higher efficiency compared with traditional scheduling algorithms without considering of NBTI aging effect.

Reference [23] presents a hierarchical two-level approach that solves a multi-objective optimization (i.e. energy consumption minimization and makespan minimization) problem. Reference [63] solves the multi-objective problem by developing a two-level schema: in the higher-level, the authors apply a heuristic approach to map jobs between clusters; in the lower-level, specific scheduling strategies are used for task scheduling locally within each cluster. Reference [103] optimizes the makespan of DAG applications by setting up a priority queue and duplicating specific tasks. In reference [49], a Minimum Energy Under Probability Constraints (MEUPC) algorithm is designed to achieve task-to-core mapping and a Trading Energy For Time (TEFT) strategy is developed to achieve task parallelism. The main goal of reference [49] is to minimize the energy consumption while satisfy the task deadline constraint.

However, in these algorithms, the tasks scheduled to the same node are assumed to run in sequence. None of the above work assumes the time-sharing execution when making the scheduling decisions. Our studies show that when the tasks allocated to the same node are run in the time-sharing manner, the finish times of individual tasks may be different from those in the sequential execution and consequently affect the makespan. Therefore, if the existing DAG scheduling algorithms are applied directly in the distributed scheduling, the actual performance of the DAG execution, no matter in terms of makespan or other objectives such as energy consumption, may not be as optimal as these scheduling methods assume.

2.3 Performance Prediction of Task Co-running in Multi-core Computers

The third work of this thesis is to perform contention-aware prediction for performance impact of task co-running in multi-core computers. In this section, we first investigate the influential factors that impact on the performance when the tasks are co-running on multi-core computers. The related work is discussed in subsection 2.3.1. Then, we present some existing performance prediction approaches in subsection 2.3.2.

2.3.1 Resource Contention Problem of Co-running Tasks

Several works have explored the performance degradation problem of co-running tasks. Survey [117] focuses on the approaches that address the shared resource contention problem of task scheduling on Chip Multi-core Processors (CMPs). Performance and energy model are built to analyze and predict the performance impact [53] [82] [83] [102] [58] [12] [115] and [67]. Reference [11] studies the impact of L2 cache sharing on the concurrent threads; Reference [115] proposes an interference model, which considers the time-variant inter-dependency among different levels of resource interference to predict the application QoS. Reference [112] decomposes parallel runtime into compute, synchronization and steal time, and uses the runtime breakdown to measure program progress and identify the execution inefficiency under interference (in virtual machine environment). Reference [3] reveals that the cross-application interference problem is related to the amount of simultaneous access to several shared resources. Based on this discovery, it proposes a multivariate and quantitative model, which has an ability of predicting cross-application interference level by considering a set of features, for example, the amount of concurrent accesses to SLLC, DRAM and virtual network, and the similarity between the amount of those accesses in virtual environments. Reference [45] predicts the execution time of an application workload for a hypothetical change of configuration on the number of CPU cores of the hosting VM. Reference [26] gained the insight into the principle of enriching the capability of the existing approaches to predicting the performance of multi-core systems. Reference [25] develops an efficient ELM based on the Spark framework (SELM), which includes three parallel subalgorithms, is proposed for big data classification.

However, most of above studies only consider part of features that may affect the co-running performance. Our experimental results reveal that there are at least 15 performance events that can affect the co-running performance.

We collect all these performance events during executing and use the feature selection methods to reduce dimensionality.

Reference [111] observed that the performance degradation of an application can be represented as a piecewise predictor function of the aggregate pressures on shared resources from all cores. Based on this observation, the author proposes to adopt the regression analysis to build a predictor function for an application. The two features that the author considered are cache and bandwidth. However, to predict the execution time and the impact of time-sharing execution on task performance, more performance events have to be considered to build the prediction model.

2.3.2 Performance Prediction Approaches

Predicting the performance (such as execution time) of co-running tasks accurately is necessary for task scheduling. When scheduling tasks with time constraints (i.e. deadlines), the worst-case execution time (WCET) is often used to indicate the upper bound of the execution time. Several works have analyzed and estimated the WCET of the tasks [34] [5] [64] [59] [29] [97]. Survey [97] suggests that the WCETs problem is hard if the supporting architecture has the components such as caches, pipelines, branch prediction, and other speculative components. The paper discusses and compares different approaches to solve the problem mentioned above.

Furthermore, estimating the execution time of co-running tasks in multi-core situations is chanllenging as we need to take the resource contention into consideration [33] [14] [56] [74]. Reference [88] builts Ernest, a performance prediction framework for large scale analytics achieves a low prediction error while having a training overhead of less than 5% for long-running jobs. Survey [98] investigates the prediction approaches that can estimate the performance of distributed tasks. Two scenarios are taken into consider in this survey: 1) a single task executing on a single node; 2) a batch of tasks co-running on a set of nodes (i.e. high performance computing system). Reference [30] presents a practical prediction model for estimating performance degradation due to shared cache. Reference [18] proposes an approach that can automatically characterize workflow requirements such as I/O, runtime, memory usage, and CPU utilization. The input data of this approach is the size of the input files. Reference [77] makes a trade-off between prediction accuracy and measurement cost. It adapts two widely used sampling strategies for performance prediction and develops a new heuristic based on feature frequencies. Reference [10] proposes an analytic modeling approach based on the use of Markovian Agents and Mean Field Analysis that can accurately represent different concurrent Big Data applications. Reference [32] aims at a formal definition of timing compositionality. It highlights challenges and suggests unsolved problems that arise in the context of compositional analyses. Reference [42] divide the resource interference of co-running VMs into two metrics, sensitivity and pressure. Sensitivity denotes how an application's performance is affected by its co-run applications, and pressure measures how it impacts the performance of its co-run applications. Further, a regression model is built to predict the two metrics with high accuracy.

Reference [45] proposes NICBLE to predict the execution time of an application for a hypothetical change of configuration on the number of CPU cores of the VM.

Reference [69] characterizes the task executions of workflow on the cloud by using a set of parameters that reflect workflow input data, VM type on which the task is executed, and hardware-dependent runtime information; then a novel fully automatic two-stage approach is developed to predict task execution times for varying input data across different cloud providers evaluated for various real-world workflows applications.

Reference [75] proposes the ProcessorMemory (ProcMem) model, which dynamically predicts the distinct task execution times depending on the implemented processor frequencies.

Reference [109] presents a framework for creating a lightweight thermal prediction system suitable for run-time management decisions. The author uses feature selection algorithms to improve the performance of previously designed machine learning methods. In addition, alternative methods are developed using neural network and linear regression-based methods to perform a comprehensive comparative study of prediction methods. Other papers such as [78] also make contributions in either performance modeling or energy prediction using performance events.

CHAPTER 3

Power-aware Scheduling Mechanisms for Virtualized Environments

The virtualization technique provides the key supporting environments for a cloud system. Nowadays, energy consumption in such a system is a critical metric to measure the sustainability and eco-friendliness of the system. This chapter develops three power-aware scheduling strategies in virtualized systems managed by Xen, which is a popular virtualization technique. These three strategies are the Least performance Loss Scheduling strategy, the No performance Loss Scheduling strategy, and the Best Frequency Match scheduling strategy. These power-aware strategies are developed by identifying the limitation of Xen in scaling the CPU frequency and aim to reduce the energy waste without sacrificing the jobs running performance in the computing systems virtualized by Xen. Least performance Loss Scheduling works by re-arranging the execution order of the virtual machines (VMs). No performance Loss Scheduling works by setting a proper initial CPU frequency for running the VMs. Best Frequency Match reduces energy waste and performance loss by allowing the VMs to jump the queue so that the VM that is put into execution best matches the current CPU frequency. Scheduling on both single core and multi-core processors is considered in this chapter. The evaluation experiments have been conducted, and the results show that compared with the original scheduling strategy in Xen, the developed power-aware scheduling algorithm is able to reduce energy consumption without reducing the performance of the jobs running in Xen.

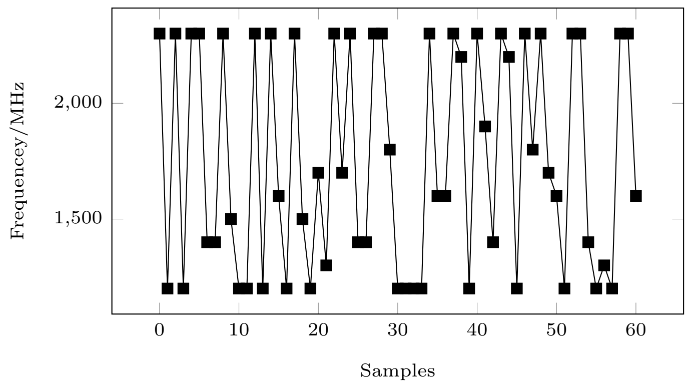


Figure 3.1: Frequency sampling of Domain 0 for 60 times; each sampling interval is 300 ms

3.1 A Motivating Example

As introduced in Chapter 1, under the OnDemand governor, DVFS is used to adjust the CPU frequency on demand. We ran an experiment on a quad-core machine to record the frequency of the CPU core, which Domain0 runs on. We pinned two vcpus of Domain0 to core 3 and recorded the frequency of the core once every 300 milliseconds (ms) for 60 times. The results are shown in Fig. 3.1. It can be seen that Domain0 does not always run with the highest frequency of 2301 MHz. Indeed, under the Ondemand governor, different tasks may run with different execution frequencies. For example, Fio, which is a I/O benchmarking tool, ran at the lowest scaling frequency (1200 MHz), while the computation-intensive benchmark BT (a benchmark application in the NAS Parallel benchmark) ran at the highest frequency, 2301 MHz.

The following illustrates the problem of energy waste and performance loss caused by the limitation of Xen in changing the CPU frequency. Fig. 3.2 shows the changes of Power-states (P-states) when four VMs (VM1-VM4) are running on a single core. The x axis is the elapsed time, while the y axis is the Power-state of the core at the corresponding time point.. The time slice of a VM (namely the time duration for which a VM runs continuously before the Xen hypervisor jumps in and schedule another VM into execution.) is 30 ms by default. The running order of the VMs in the experiment is also labelled in the figure. The power governor checks and changes, if necessary, the CPU frequency every 10 ms by at most one level [54]. Assume that the P-states demanded by VM1, VM2, VM3 and VM4 are P1, P9, P3 and P7, respectively. In this example, the initial P-state is set to be P3. When the current frequency deviates from (higher or lower than) the best frequency of a running VM, the power governor adjusts the frequency towards the best frequency. However, it can only adjust the frequency by one level every 10 ms due to the feature of Xen in frequency changing. With this restriction, the actual CPU frequencies over time are those highlighted by the bold black line. When a VM runs on a frequency higher or lower than its best execution frequency, Energy waste or performance loss occurs. The difference between the best frequency and the actual frequency represents the amount of energy waste or performance loss. In this figure, the red area and the shadowed grey area represent the amount of energy waste and performance loss, respectively. For example, at time 0, the VM (VM4) only requires P5 (the best P-state), while the current actual frequency is P3 (initial P-state). Since the current P-state is higher than the best P-states of the VM, the frequency is adjusted down by one level every 10 ms until it reaches the best frequency or the VM is scheduled out after its time slice of 30 ms is used up. In the case of VM4, the VM is scheduled out before the actual frequency reaches the best frequency. VM3 is scheduled in after VM4. Since the best frequency of VM3 is P3, which is higher than the current running frequency, the frequency is adjusted up. Since the frequency can only be adjusted by one level every 10 ms, VM3 still runs at a frequency lower than its best frequency in the first 10 ms of VM3's time slice, which leads to the performance loss (indicated by the shadowed grey area) and will consequently increase the execution time of the application that is running in the VM. In the remaining time slice, VM3 runs perfectly at its best frequency.

The above example suggests that the feature of Xen in adjusting the CPU frequency may cause both energy waste and performance loss. The objective of this work aims to improve the situation. We adapt the default SEDF scheduling

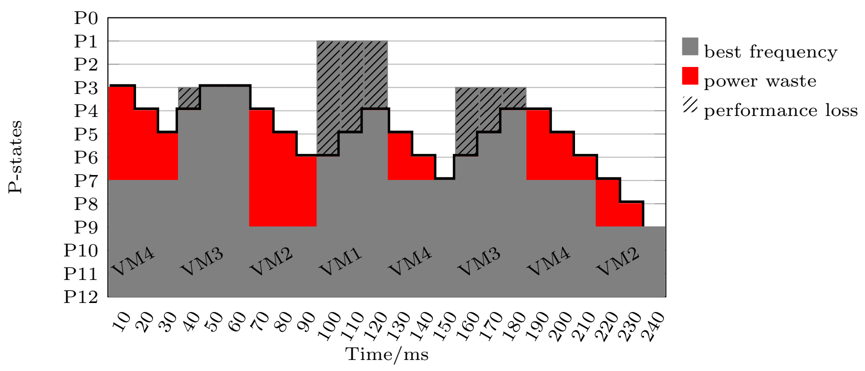


Figure 3.2: Energy waste and performance loss under DVFS in Xen

policy so as to minimize the energy waste and performance loss under DVFS.

3.2 Performance and Energy Model for the DVFS-enabled Xen

3.2.1 Performance Model

Assume a set of independent tasks T = $\{T_1, T_2, ..., T_n\}$. Task T_i runs in VM_i. f_i denotes the best frequency of VM_i. t_i denotes the execution time of task T_i when VM_i runs at the frequency of f_i (T_i is executed in VM_i). P_s denotes the scheduling time slice, which is 30 ms by default, while P_f denotes the frequency scaling slice, which is 10 ms. Let $c(f_i')$ denote the equivalent execution rate of T_i when VM_i runs at the frequency f_i'. $c(f_i')$ can be calculated by Equation 3.1, where $F_t(f_i')$ is the function of execution time over CPU frequency.

$$c(f_i') = \frac{t_i}{F_t(f_i')} \tag{3.1}$$

$f_i'(j)$ denotes the frequency which task T_i runs at in the jth time interval. $c(f_i'(j))$ denotes the execution rate of T_i in the jth time interval. Inequality 3.2 can be used to determine the number of intervals that VM_i uses to complete the execution of task T_i, which is the minimal value of m that satisfies Inequality

3.2.

$$\sum_{j=0}^{m} c(f_i'(j))P_f \ \geq t_i \tag{3.2}$$

Then the total execution time of task T_i when it is not always running at its best frequency f_i (denoted by t_i') can be modelled as Equation 3.3, where m is the minimal value that satisfies Inequality 3.2.

$$t_i' = \sum_{j=0}^{m} P_f \tag{3.3}$$

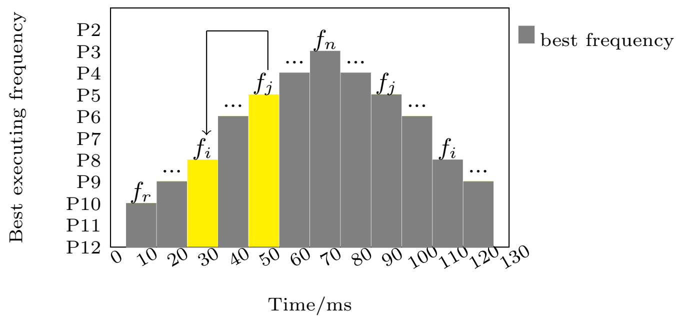
3.2.2 Power Consumption Model

When task T_i runs at the frequency of f_i', Equation 3.4 is the classic equation to calculate the power consumption rate of CPU for running T_i [61] (denoted by r_i), where C is the capacitance being switched per clock cycle, V is the voltage, A is the activity factor indicating the average number of switching events undergone by the transistors in the chip and f_i' is the frequency.

$$r_i(f_i') = A \times C \times V^2 \times f_i' \tag{3.4}$$

Inequality 3.2 is used to determine the number of intervals that VM_i uses to complete the execution of task T_i. The total power consumption of task T_i, denoted by e_i, can be modelled by Equation 3.5, where $f_i'(j)$ is the frequency which task T_i runs at in the jth time interval, same as in Equation 3.2.

$$e_i = \sum_{j=0}^{m} r_i(f_i'(j))P_f \tag{3.5}$$



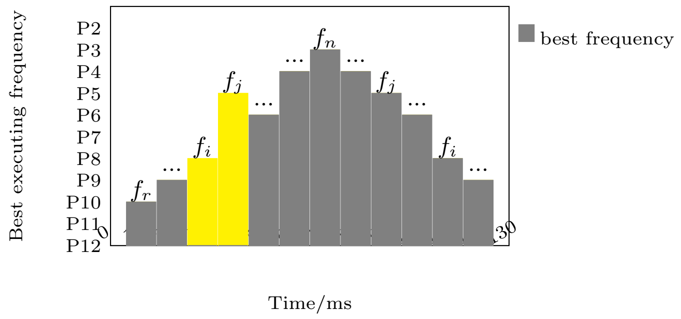


Figure 3.3: "Least Performance Loss" Scheduling Strategy (above); Illustration of changing the execution position of a randomly selected task (below)

3.3 Scheduling Strategies

3.3.1 The Scheduling Strategy with Least Performance Loss

The power management in Xen can only adjust the power state by at most one level every 10 ms. The frequency gap between the current CPU frequency f and the task T_i's best (desired) executing frequency f_i will lead to either performance loss or energy waste. When the current CPU frequency f is lower (or higher) than T_i's best executing frequency, f_i, and the power management cannot increase (or reduce) f to f_i immediately, performance loss (or energy waste) occurs.

Given the current CPU frequency and a set of tasks, Theorem 1 gives the "Least Performance Loss" Scheduling strategy (LLS), namely the execution order of the tasks that leads to the least performance loss.

Theorem 1. *Given a set of tasks, $T = \{ T_1, T_2, \ldots, T_n\}$, the best CPU frequency of T_i is f_i and the set of tasks are run in a time-sharing manner. Assume $f_1 \leq f_2 \leq \ldots \leq f_n$. If the current CPU frequency is f, then given the current CPU frequency, the LLS strategy (i.e., the execution order that leads to the least performance loss for the set of tasks) is to run the tasks in the following order, where T_r's frequency f_r is the highest frequency that is less than the current frequency f.*

$T_r, T_{r+1}, \ldots, T_{n-1}, T_n, T_{n-1}, \ldots, T_{r+1}, T_r, T_{r-1}, \ldots, T_2, T_1,\ T_2, \ldots,\ T_n,\ T_{n-1}, \ldots,\ T_1$ $\ldots$

Namely, the execution order is to start from T_r, go up to T_n in the increasing order of frequency and then come down to T_1 in the decreasing order of frequency, and that the upward and downward execution pattern in terms of frequency repeat until all tasks have been completed.

Proof. The performance loss is related to the frequency gap of the tasks during the execution. Performance loss increases with the increase in the frequency gap. We prove this theorem by proving that any change in the execution order from the LLS strategy will lead to the increase of the frequency gap, thus the performance loss.

We randomly change the execution position (specified by the LLS strategy) of a randomly selected task. Assume task T_j is moved to the position after task T_i. Without the loss of generality, we assume $j > i + 1$, (i.e., we move T_j forwards as shown in Fig. 3.3.

Before the change, the frequency gap among the relevant tasks (i.e. task T_i, T_{i+1}, T_{j-1}, T_j, T_{j+1}) is:

$E = (f_{i+1} - f_i) + (f_j - f_{j-1}) + (f_{j+1} - f_j)$

$\quad = f_{i+1} - f_i - f_{j-1} + f_{j+1}$.

After the change, the frequency gap among the involved tasks is:

$E' = (f_j - f_i) + (f_{j+1} - f_{j-1})$.

Note that the gap of $f_j - f_{i+1}$ is not counted in the expression above since it does not cause performance loss (but energy waste) even if Xen cannot adjust the frequency timely from f_j to f_{i+1}.

The difference between E' and E is:

$$E' - E = (f_j - f_i) + (f_{j+1} - f_{j-1}) - (f_{i+1} - f_i - f_{j-1} + f_{j+1})$$
$$= f_j - f_{i+1}.$$

Since $f_j \geq f_{i+1}$, we get $E' \geq E$.

In the similar way, we can prove the theorem also holds when $i > j$ (i.e., moving the task backwards). Therefore, given the current CPU frequency, the LLS strategy generates the least performance loss for a set of tasks.

□

3.3.2 The Scheduling Strategy with No Performance Loss

The previous section derives LLS, the scheduling strategy with the least performance loss, given the current CPU frequency. LLS requires re-arranging the execution order of the VMs. In this section, we will derive the scheduling strategy under which there is no performance loss for a set of tasks. We call this strategy the No performance Loss Scheduling (NLS) Strategy. NLS aims to ensure all tasks run with the frequencies no less than their best frequencies. NLS does not reorder the VMs' execution. The VMs can be executed in the order of their positions in the run-queue. Rather, NLS calculates the initial CPU frequency that the CPU needs to be set with in order for all VMs to run without performance loss.

According to the Xen power management policy, the execution frequency can be modified once every 10 ms, which we call the *frequency scaling slice*. The default time slice, which we call the scheduling slice, for running a VM in Xen is 30 ms. After 30 ms, the Xen hypervisor jumps in and schedule another VM to run. Therefore, the frequency can be changed three times at most in each scheduling slice. As shown in Fig. 3.4, $f_k(j)$,$f_k(j+1)$,$f_k(j+2)$ indicates the three execution frequencies of task T_k in the three frequency scaling slices (indexed as j, $j+1$ and $j+2$ in the example of Fig. 3.4) in T_k's scheduling slice. To ensure that task T_k can execute with at least its best frequency, the frequency of the frequency scaling slice before $f_k(j)$ (i.e., the $(j-1)$-th frequency scaling slice) should be at least $f_k(j) - \Delta f$, where Δf is the frequency that can be changed at most each frequency scaling slice (which is 100 MHz, i.e., one P-state, in Xen). $(j-1)$-th frequency scaling slices falls in the scheduling slice of task T_{k-1}, which means that the execution frequency that T_{k-1} has to run with, denoted by $f_{k-1}(j-1)$, has to be $(f_k(j) - \Delta f)$ even if T_{k-1} does not need such a high running frequency. The best frequency of task T_{k-1}, i.e., f_{k-1} is shown by the yellow bar, which is lower than $f_{k-1}(j-1)$. Similarly, $f_{k-1}(j-2)$ has to be $f_{k-1}(j-1) - \Delta f$ or f_{k-1} (T_{k-1}'s best frequency), whichever is higher.

In general, assuming that T_k has the highest best frequency in the set of tasks (i.e., f_k is highest) and that j, $j+1$ and $j+2$ are the three frequency scaling slices in T_k's first scheduling slice during the running of the set of tasks, Equation 3.6 can be used recursively, starting from $i = j$, to calculate the running frequency in each frequency scaling slice before j-th frequency scaling slice (it is obvious that $f_k(j)$, $f_k(j+1)$ and $f_k(j+2)$ should all be f_k), so that all VMs can run without performance loss, i.e., with the frequencies no less than their corresponding best frequencies.

The algorithm for performing the recursive calculation is outlined in Algorithm 1. The output of Algorithm 1 is the value of $f_1(1)$, i.e., the starting frequency that the CPU has to be set with in order for the set of tasks to run without performance loss.

Note that although NLS guarantees no performance loss, it may cause energy waste. The shadowed areas above the coloured bars in Fig. 3.4 represent the energy waste.

$$f_{k'}(i-1) = max(f_k(i) - \Delta f, f_{k'}) \tag{3.6}$$

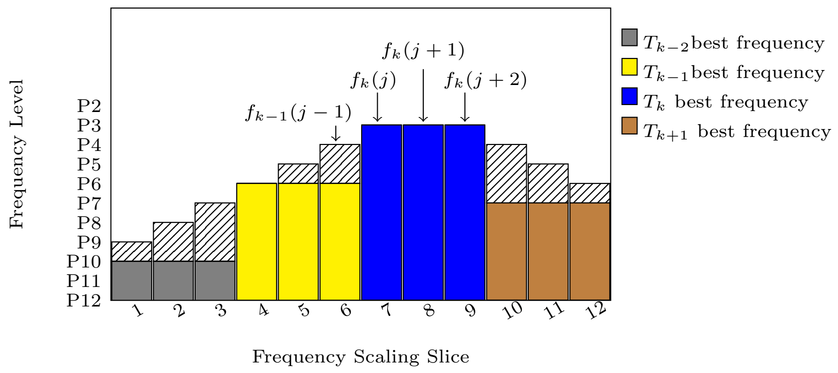


Figure 3.4: "No Performance Loss" Scheduling Strategy

where,

$$k' = \begin{cases} k & \text{if } (i-1)\text{-th frequency scaling slice is in } T_k\text{'s scheduling slice} \\ k-1 & \text{if } (i-1)\text{-th frequency scaling slice is in } T_{k-1}\text{'s scheduling slice} \end{cases} \tag{3.7}$$

Algorithm 1 No performance loss scheduling strategy

Input: Tasks T_1,T_2,...,T_n in the run-queue, whose best running frequencies are f_1,f_2,...,f_n; T_K is the task with the highest best frequency f_K; j is the index of the first frequency scaling slice in T_K's scheduling slice

Output: $f_1(1)$

```
1 $k = K, f_k(j) = f_K$; for $(i = j; i \geq 2; i--)$ do
2     if $(j-1)th$ frequency scaling slice is in $T_k$'s scheduling slice then
3         $k' = k$;
4     else
5         $k' = k - 1$;
6     $f_{k'}(i-1) = max(f_k(i) - \Delta f, f_{k'})$;
```

3.4 BFM Scheduler

In section 3.3, we presented two scheduling strategies: The Least Performance Loss Scheduling Strategy (LLS) and No Performance Loss Scheduling Strategy (NLS).

NLS will achieve the shortest execution time since every VM will execute with a frequency equal to or higher than its best frequency. Those VMs which run with a frequency higher than its best frequency (in order to guarantee that other VMs can run with their best frequencies) will cause energy waste. Therefore, NLS is designed with maximizing the performance as the only goal. In NLS, we propose the method to determine the minimal initial frequency that the CPU has to be set with in order to guarantee that no VM will experience performance loss.

Different from NLS, LLS does not reeve up CPU frequency to guarantee there is no performance loss, but makes the best effort to reduce the performance loss by manipulating the VMs' execution order. LLS does not artificially set the initial CPU frequency for running a set of VMs, but goes along with the current CPU frequency.

In order to guarantee that every VM's deadline is met, however, the SEDF scheduler in Xen requires the VMs to be run in the order of deadline (earliest deadline first). In this execution environment, meeting the deadlines is the top priority and VMs' execution order may not be able to be adjusted in the way designated by LLS. Thus, in this section we present a power-aware SEDF scheduling strategy, called the Best Frequency Match strategy (BFM). BFM aims to make the best effort to reduce performance loss subject to respecting the principle of SEDF, i.e., meeting all VMs' deadlines.

Next, we first present BFM for single-core processors and then extend it to multi-core processors.

3.4.1 BFM for Single-core Processors

BFM aims to minimize the performance loss while satisfying the VMs' CPU requirement specified in SEDF.

Assume that a set of VMs T_i ($1 \leq i \leq n$) are in the run queue of a single core with their CPU requirement expressed as (p_i, s_i, x_i) and that task T_i's best execution frequencies is f_i. The deadline of each VM is recalculated when the current scheduling slice is finished. BFM checks the deadline and the best frequency of each VM (VCPU) in the run-queue of the CPU core. If the first

VM in the run-queue (i.e., the one with the earliest deadline) has the smallest gap between its the best frequency and the current executing frequency, the VM will be scheduled. However, if there are other VMs in the queue which have smaller frequency gaps than the first VM, scheduling the first VM (with the earliest deadline) will cause either performance loss (if the current frequency is less than the best frequency) or energy waste (if the current frequency is higher than the best frequency) compared with scheduling a VM with smaller frequency gaps. Under this circumstance, BFM checks if there is any better scheduling choice in the following way.

Firstly, BFM identifies the VM, for instance T_j, whose best frequency has the smallest gap with the current CPU frequency. Before allowing task T_j to jump the queue, BFM needs to make sure that all the VMs queueing before T_j satisfy Inequality 3.8, where t_c is the current time while L_i is the position of task T_i in the run-queue. Inequality 3.8 can be understood in the following way. The scheduling slice of a VM is P_s. Task T_i, whose position in the original queue is L_i, needs to wait $(L_i - 1) \times P_s$ for T_i being put into execution. After the queue jump, the waiting time of T_i becomes $L_i \times P_s$. The waiting time plus the VM's running duration, which is P_s, must be no greater than VM_i's deadline d_i, which results in Inequality 3.8.

$$\forall L_i < L_j \quad d_i - [t_c + (L_i + 1) * P_s] \geq 0 \tag{3.8}$$

If Inequality 3.8 can be satisfied for all the VMs before T_j, BFM allows task T_j to jump the queue. If not, BFM continues to find the VM which has the second smallest gap between its best frequency and the current frequency, and applies Inequality 3.8 to determine whether the queue-jumping is allowed. The process repeats until BFM finds a VM that is eligible to jump the queue or all VMs have been considered (in this case, no VMs can jump the queue and BFM schedules and run the first VM in the queue, same as the SEDF scheduler).

The pseudo code of BFM on a single core is presented in Algorithm 2.

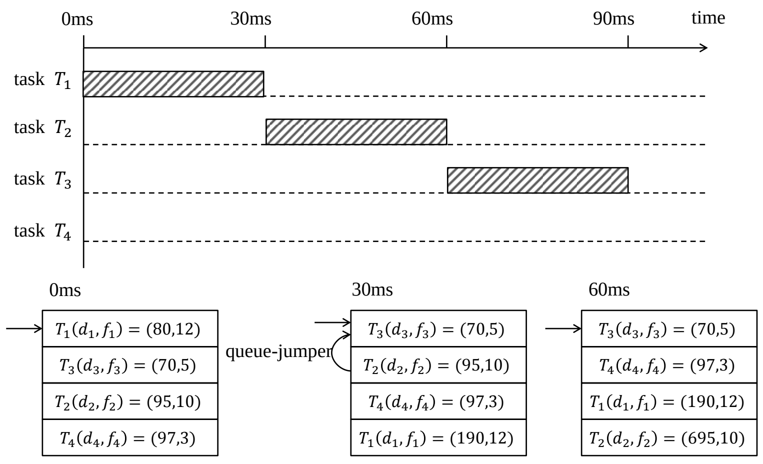


Figure 3.5: An example of the BFM scheduling strategy

An example is used in Fig. 3.5 to illustrate the working of BFM. 4 VMs are considered in this example: $T_1(20, 100)$, $T_2(5, 100)$, $T_3(10, 100)$, $T_4(30, 100)$, where the first number of the pair is the time that a VM has to run in a period, which is indicated by the second number. For example, T_1 has to run at least 20 ms in every period of 100 ms. In the beginning of each time slice, all VMs are sorted by their deadlines in the run queue. d_i presents the deadline of T_i and f_i is its best frequency of T_i. At the time point of 30 ms, T_3 has the earliest deadline, while T_2 has the smallest frequency gap with the current CPU frequency P12. Under this circumstance, BFM will check if scheduling T_3 first (i.e., allowing T_2 to jump the queue) will cause the VMs before T_2 (i.e., T_3 in this example) to miss the deadlines. In this case, it will not and therefore T_2 jumps the queue successfully. At 60 ms, T_1 has the the smallest frequency gap with the current frequency. However, it is rejected for T_1 to jump the queue, since otherwise T_3, the VM before T_1 in the queue, would miss its deadline.

Algorithm 2 BFMS scheduling Algorithm

```
Input: A set of VMs, $T_i$ $(1 \leq i \leq n)$, with their CPU requirements $(p_i, s_i, x_i)$;
       the best frequency of $T_i$, $f_i$; scheduling slice $P_s$; frequency scaling slice
       $P_f$; current time $t_c$; $T_i$'s deadline $d_i$; $T_i$'s location in the queue, $L_i$;
7  for The end of each scheduling slice do
8      Calculate the deadlines of all tasks;
       Sort the tasks in the ascending order of deadline in queue $Q$;
       Obtain the VM with the earliest deadline, denoted by $T_e$;
       for All tasks in the run queue do
9          Calculate the frequency gap $g_i$ between the best frequency of task $T_i$ and
           the current frequency $f$
10     Sort the tasks' frequency gaps in the ascending order;
       Get the first frequency gap, $g_k$, in the frequency-gap sorting queue and
       denote the corresponding task by $T_k$;
       if $T_k$ is $T_e$, i.e., $g_e$ is the minimal gap then
11         Schedule $T_e$ to run
12     else
13         while $g_k$ is no more than $g_e$ do
14             if Each task $T_i$ before $T_k$ in queue $Q$ satisfies $d_i - [t_c + (L_i + 1) * P_s] \geq 0$
               then
15                 Schedule $T_k$ to run next;
16             Get the next frequency gap, $g_k$, in the frequency-gap sorting queue
               and denote the corresponding task by $T_k$;
17         Schedule $T_e$ to run next;
```

3.4.2 BFM for Multi-core Processors

In this section, we extend the BFM strategy for a single core to multi-core processors. In this section, we denote BFM for single core by BFMS and BFM for multi-core by BFMM. Compared with BFMS, the main additional work of BFMM is to allocate a set of VMs among multiple cores in the processor. This section first presents Theorem 2, which is used as the principle for allocating the VMs, and then presents a actual allocation method. After the set of VMs are allocated, the VMs are scheduled and run using BFMS in each individual core.

Theorem 2. *Assume there are n VMs and m cores in the processor (assume n can be divided by m). The following allocation method results in the least performance loss.*

The n VMs are sorted in the ascending order of their best frequencies. The sequence of the VMs are denoted by $T_1, T_1, ..., T_i, ..., T_n$ (VM T_i's best frequency is f_i). The sequence of VMs are allocated evenly into m cores. Namely, assuming j denotes the index of core $1 \leq j \leq m$, VMs $T_{(j-1)\frac{n}{m}}$ to $T_{(j)\frac{n}{m}}$ are allocated to core j. In this way, tasks with the nearest best frequency will be allocated on the same core, which will lead to the least performance loss.

Proof. We prove the theorem by proving that exchanging any two VMs between different cores in the allocation method will lead to the increase in performance loss.

Assume we exchange task T_i on core C_I with task T_j on core C_J (assume $I < J$, i.e. the VMs' best frequency on C_I are lower than those on C_J). If the total frequency gap after the exchange is higher than that before the exchange, the performance loss after the exchange must be no less than that before the exchange.

Before the exchange, the total frequency gap E between the relevant VMs is:

$$E = (f_i - f_{i-1} + f_{i+1} - f_i) + (f_j - f_{j-1} + f_{j+1} - f_j) \quad (3.9)$$

After exchanging, the total frequency gap E' between the involved tasks can be calculated as:

$$E' = f_j - f_{i-1} + f_{j+1} - f_i \tag{3.10}$$

Note that the gap of $f_j - f_{i+1}$ and $f_{j-1} - f_i$ are not counted in the expression above since it does not cause performance loss (but energy waste) even if Xen cannot adjust the frequency timely from f_j to f_{i+1} and f_{j-1} to f_i. Thus,

$$E' - E = f_j + f_{j-1} - f_i - f_{i+1} \tag{3.11}$$

Since f_j and f_{j-1} are greater than f_{i+1} and f_i, we get $E' \geq E$.

Note the above expressions for calculating E and E' capture the general cases. There are special cases where f_i and f_j are the highest or lowest frequencies in their cores. These special cases can also be proved in a similar way. $\square$

Theorem 2 essentially states the allocation principle that the VMs with close best frequencies should be allocated to the same core. In SEDF, however, a VM, T_i, has the CPU requirement, specified by the first two parameters of the triple (s_i, p_i, x_i). T_i's CPU requirement can be computed as $\frac{s_i}{p_i}$. When BFMM allocates the VMs, it needs to make sure that all VMs allocated to the same core can meet their CPU requirements. According to the schedulability analysis in the literature [108], if the sum of $\frac{s_i}{p_i}$ for a group of VMs allocated in a core is less than 100%, the CPU requirements of this group of VMs can be met by SEDF. Based on the consideration of the CPU requirement, we adjust the allocation method in Theorem 2 and present the allocation method used in BFMM. The fundamental idea of the adjusted allocation method is as follows.

When allocating a set of VMs to a set of cores, $C = \{C_1,C_2,...,C_m\}$, BFMM first sorts the VMs in the ascending order of their best frequencies. The sorted VM set is $T = \{T_1,T_2,...,T_n\}$ (i.e. T_1 has the lowest best frequency while T_n has the highest best frequency). VM T_i's best frequency and CPU requirement are f_i and $\frac{s_i}{p_i}$. BFMM allocates the VMs from T_1 to T_n one by one to m cores.

It tries to allocate the VMs with the closest best frequencies to the same core as long as the CPU requirement of the VMs on the same core can be satisfied, i.e., the sum of $\frac{s_i}{p_i}$ on the core is less than 100%. When BFMM allocates T_i and finds that T_i's CPU requirement cannot be met in the core, it moves to the next VM and check if the next VM can be allocated the core. The process continues until all VMs are examined for the current core. BFMM then moves to the next core and tries to allocate VMs to the core. This process repeats until all VMs are allocated. The allocation method for BFMM is outlined in Algorithm 3.

Algorithm 3 The VM allocation method in BFMM

```
Input: A set of VMs $\{T_i\}(1 \leq i \leq n)$; the frequency of $T_i$, $f_i$; the CPU
       requirement of $T_i$, $\frac{s_i}{p_i}$); a set of cores $C_j, (1 \leq j \leq m)$;
18 for All VMs do
19     Sort the VMs in the ascending order of frequency and obtain the sorted list
       of $\{T_1, T_2, ..., T_n\}$, i.e., $f_1 \leq f_2 \leq ... \leq f_n$;
20 The current core $C_c$ is initialized to be $C_1$, i.e., $c = 1$; for $T_1 \rightarrow T_n$ do
21     if VM $T_i$ has not been allocated then
22         Calculate the total CPU requirement of VMs allocated to $C_c$, denoted
           by $\sum_c$; if $\frac{s_i}{p_i} + \sum_c \leq 100\%$ then
23             allocate $T_i$ to core $C_c$;
24         else
25             for $j = i + 1; j \leq n; j + +;$ do
26                 if VM $T_j$ satisfies $\frac{s_j}{p_j} + \sum_c \leq 100\%$ then
27                     Allocate $T_j$ to core $C_c$;
28             $c + +;$
```

Note that when there are no deadlines, the BFM strategy essentially becomes LLS. Another point is that BFM works by re-arranging the execution order of the jobs in the run queue (i.e., allowing queue-jumping), which is designated by SEDF, only when the deadline allows. So in terms of meeting real-time requirements, BFM is as good as SEDF. The only case where BFM is unable to guarantee a tasks deadline is when SEDF is unable to meet its deadline. In this case, BFM will simply disallow the queue-jumping and the scheduling behaviour of BFM will be the same as SEDF.

3.5 Evaluation

3.5.1 Experimental Setup

We conducted the experiments on the server with Intel(R) Core(TM) i7-3615QM CPU@2.30GHz processor, 32GB RAM and 122GB hard drives. The processor has 4 physical cores and supports 12 performance states from the minimum frequency of 1200 MHz to the maximum frequency of 2301 MHz. Xen-4.4.1 hypervisor with the kernel version of 3.13.0-32-generic was used to create the virtualized system. SEDF is selected as the vCPU scheduler and the Ondemand governor as the DVFS runtime power management. Each VM in the experiments was run with 2 VCPUs and 256M memory.

The best CPU frequency for running a task in a VM is determined in the following way. We first set the Xen governor to userspace and then set the CPU frequency using the commands: xenpm set-scaling-minfreq and xenpm set-scaling-maxfreq. We run the task with different frequencies and record the execution time and energy consumption. Fig. 3.6 shows the execution times of different benchmark tasks running on different frequencies.

As we can see in Fig. 3.6, as the execution frequency increases, the decreasing trend of the execution times of all the benchmarks diminishes. The total energy consumption of completing a task can be calculated by the execution frequency times the corresponding execution time (i.e., the value on the x axis times the corresponding value on the y axis). Fig. 3.7 shows the power consumption of the benchmark tasks running on different frequencies. In this chapter, the best CPU frequency of a task is defined as the frequency which leads to the lowest energy consumption of the task. According to Fig. 3.7(a) to Fig. 3.7(d), we can know that the best frequencies of the four benchmarks, EP, LU, CG and BT, are 2300 MHz, 1400 MHz, 1200 MHz and 1700 MHz, respectively. A task's execution time when it is run with the best frequency is called the best frequency execution time.

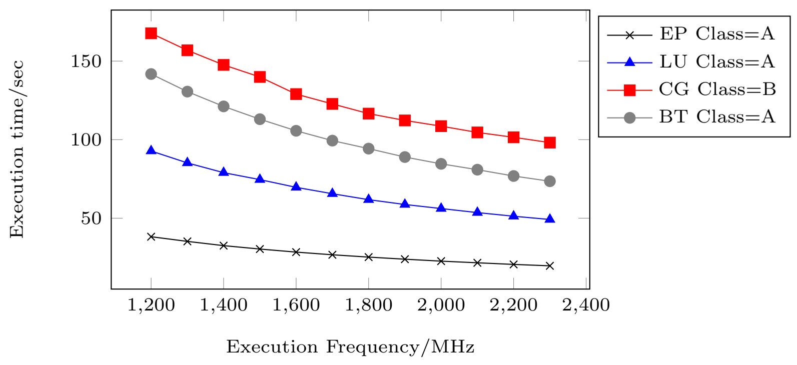
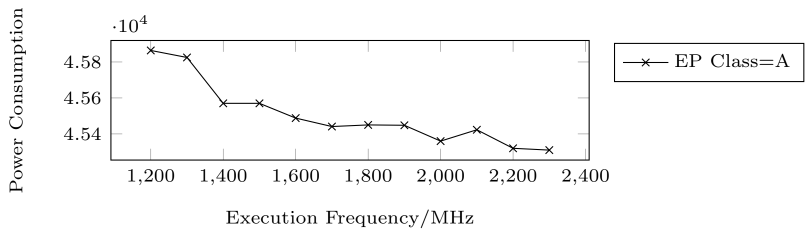


Figure 3.6: The execution times of different benchmark tasks running with different frequencies

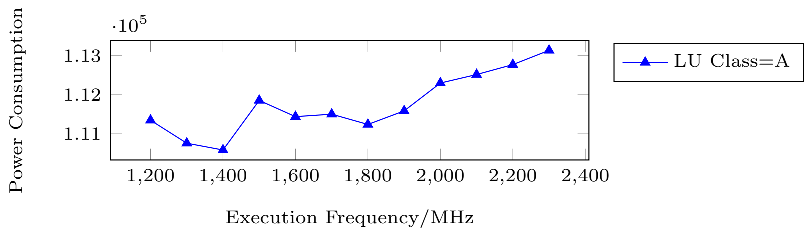
3.5.2 Experiments on Single-core Processors

In this section, we compare the BFM scheduler with the SEDF scheduler in Xen in terms of the performance of managing the VMs in a single core. Four benchmark applications, EP, LU, CG and BT, are used, which are denoted T_1, T_2, T_3 and T_4, respectively. The best frequencies of T_1 to T_4 are 2300 MHz, 1400 MHz, 1200 MHz, 1700 MHz. Their best frequency execution times are 9850 ms, 7899 ms, 16768 ms, 9938 ms. We run these tasks, each in a separate VM, on a single core under the SEDF and the BFM schedulers. Fig. 3.8 compares the execution times of the tasks under these two schedulers. The best frequency execution time is also depicted in the figure for comparison.

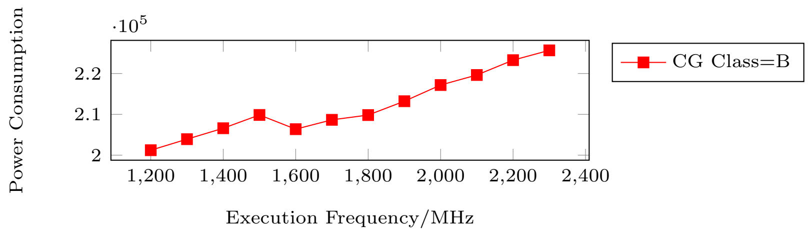
Our experimental records for Fig. 3.8 show that the execution times of $T1$, $T3$ and $T4$ are reduced by 10 ms, 140 ms and 10 ms, respectively, under BFM, compared to SEDF. This can be explained as follows. Under SEDF, the tasks, especially those frequency-sensitive tasks (i.e. the executing frequency has a big influence on its execution time, for instance T_3), may run with the frequency which is lower than its best frequency, and therefore the execution time may decrease. For the tasks with high best frequencies (i.e. T_1 and T_4), the execution times under SEDF increase, comparing to their best frequency execution times. This is because they may be scheduled behind some tasks with low best frequencies and Xen cannot adjust the frequency up timely during the



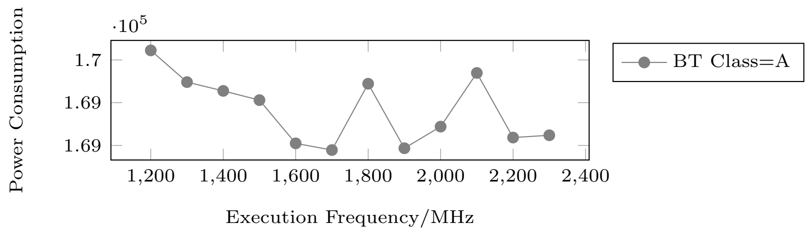
(a) EP



(b) LU



(c) CG



(d) BT

Figure 3.7: The power consumption of benchmarks running with different frequencies

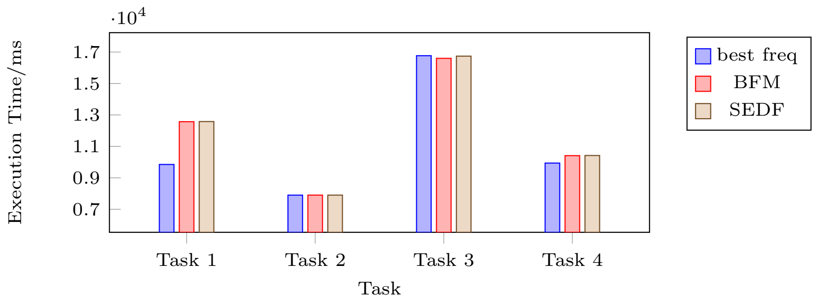


Figure 3.8: Execution times of tasks on a single core under SEDF and BFM

scheduling process. On the countrary, BFM may allow other tasks to jump the queue if they have smaller gap between the best frequency and the current frequency (subject to the deadline requirement) and therefore give Xen more time to build up the frequency to run the high frequency tasks.

Fig. 3.9 compares the power consumption of the tasks running under SEDF and BFM. The power consumption of the tasks running with their best frequencies, i.e., the frequencies that result in the minimal power consumption by tasks, are also drawn in the figure for comparison. It can be seen that the power consumption under BFM is much less than that under SEDF for the tasks T_2 and T_3. This is because under SEDF, the tasks, especially those with low best frequencies, may run with the frequency higher than what they need (the best frequency), which causes energy waste. High best frequency tasks, for example, T_1 (EP), consume 2.2655×10^7, 2.2728×10^7, 2.2720×10^7, respectively, with the best frequency, under SEDF and under BFM. SEDF scheduler leads to a power waste of 73000 (i.e. 2.2728×10^7 - 2.2655×10^7) while BFM leads to a power waste of only 65000. These results suggest that BFM can reduce energy consumption while improving performance by allowing the suitable VMs to jump the queue to fill in the gap between the current frequency and the best frequency of the VM at the head of the queue under SEDF.

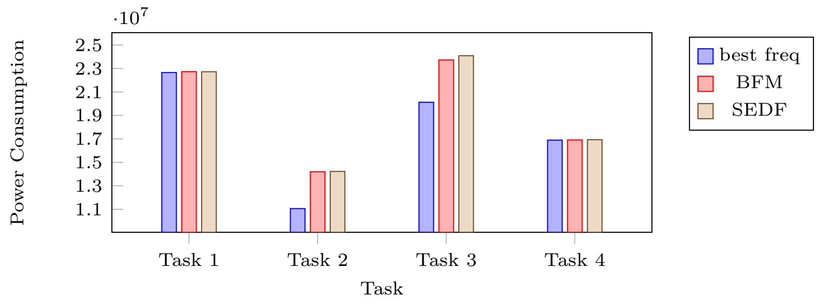


Figure 3.9: Power consumption of tasks on a single core under SEDF and BFM

3.5.3 Experiments on Multi-core Processors

We use SEDF and BFMM to schedule and run 20 tasks on a DVFS-enabled Quad-Core processor. Each task T_i has the best execution frequency and the CPU requirement, represented by a tuple $(f_i, \frac{s_i}{p_i})$. We set the tasks' execution times so that every task's execution time is 20000 ms when they are run with their best frequencies.

Fig. 3.10 and Fig. 3.11 show the allocation of 20 tasks on the quad-core processor under SEDF and BFMM. The results show that BFMM allocates the VMs with closer frequencies to the same core, compared with SEDF. Fig. 3.12 and Fig. 3.13 show the performance and power consumption of these 20 tasks, respectively. As can be seen from Fig. 3.12, the VMs with high best frequencies (e.g., T_{13} and T_{14}) have much longer execution times under SEDF than under BFMM. The reason is similar as the reason for the performance gap shown in Fig. 3.8. Namely, under SEDF these tasks with high best frequencies may be scheduled to run behind the tasks with low best frequencies and therefore Xen cannot adjust the frequency up timely. AS we can see from Fig. 3.13, BFMM reduces power consumption of all tasks. The reason for this is also similar as the reason for the difference of the power consumption in Fig. 3.9.

Core 1	***Core 2***	***Core 3***	***Core 4***
T_1(1200,20%)	T_2(1700,15%)	T_3(1400,5%)	T_4(2000,15%)
T_5(2000,10%)	T_6(1300,15%)	T_7(2200,35%)	T_8(2100,20%)
T_9(1300,20%)	T_{10}(1600,20%)	T_{11}(1800,20%)	T_{12}(1900,25%)
T_{13}(2100,20%)	T_{14}(2200,25%)	T_{15}(2300,20%)	T_{16}(1300,5%)
T_{17}(1200,25%)	T_{18}(1400,15%)	T_{19}(1900,15%)	T_{20}(2300,25%)

Figure 3.10: Task consolidation on Quad-Core using SEDF scheduler

Core1	***Core2***	***Core3***	***Core4***
T_1(1200,20%)	T_3(1400,5%)	T_4(2000,15%)	T_{14}(2200,25%)
T_{17}(1200,25%)	T_{10}(1600,20%)	T_5(2000,10%)	T_{20}(2300,25%)
T_9(1300,20%)	T_2(1700,15%)	T_8(2100,20%)	T_{15}(2300,20%)
T_6(1300,15%)	T_{11}(1800,20%)	T_{13}(2100,20%)	
T_{16}(1300,5%)	T_{12}(1900,25%)	T_7(2200,35%)	
T_{18}(1400,15%)	T_{19}(1900,15%)		

Figure 3.11: Task consolidation on Quad-Core using BFM scheduler

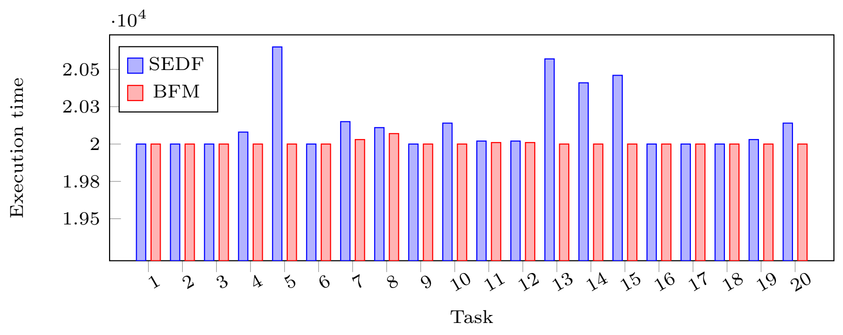


Figure 3.12: Execution times of tasks on a multi-core processor under SEDF and BFMM

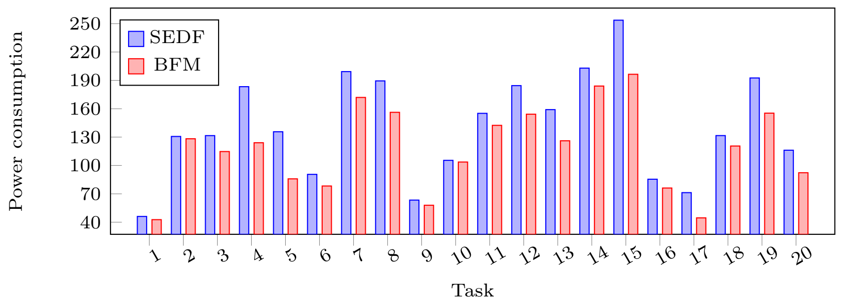


Figure 3.13: Power consumption of tasks on a multi-core processor under SEDF and BFMM

3.6 Summary

This work reveals that the traditional scheduling strategies in virtualized systems managed by Xen may lead to performance loss and energy waste, due to the limitation of Xen in adjusting CPU frequency, i.e., Xen can only check and change the CPU frequency at most once every 10ms. This chapter presents four scheduling strategies to remedy this situation, which are the Least performance Loss Scheduling (LLS) strategy, the No performance Loss Scheduling (NLS) strategy, the Best Frequency Match strategy for a single core (BFMS) and the Best Frequency Match strategy for multiple cores (BFMM). These strategies make use of the scheduling behaviour in the Xen hypervisor and aim to reduce energy consumption while mitigating performance loss. The effectiveness of these strategies is theoretically proved and also evaluated by the experiments. The philosophy used in BFM to reduce performance loss and energy consumption may also be applied to other schedulers in Xen, such as the Credit scheduler.

To make scheduling decisions, particularly for the tasks with time constraints, it is often assumed that the execution time of a task is known in advance. In some studies, the execution time of a task is estimated by the-number-of-instructions/processing-capacity. However, when a task is co-running with other tasks on a multi-core processor, it becomes much more complex to estimate its execution time. Given a schedule (i.e. execution order of the tasks and the allocation of the tasks to CPU cores), most existing work assumes that the tasks scheduled on the same computing node run in sequence. In reality, however, the tasks are often run in the time sharing manner, which leads to inaccurate estimation of the execution time of the tasks. The next chapter proposes the method to estimate the execution time of a co-running task and consequently to model the makespan of the tasks that are running in the time-sharing manner. The next chapter starts with a simple scenario where Bag of Tasks (BoT) are scheduled, and then extends to a more complex scenario: DAG tasks are scheduled and run in the time-sharing manner. Based on the constructed makespan models, the scheduling strategies are further developed for the DAG tasks under the time-sharing execution.

CHAPTER 4
Scheduling DAG Applications for Time Sharing Systems

Accurately modeling the makespan of a task is really important for task scheduling strategies. In order to satisfy each task's time constraint (i.e. deadline), the worst-case execution time of a task is taken into consideration when making scheduling decisions. When computing the makespan of a task that is co-running with other tasks on multi-core processors, it is typically assumed that the tasks scheduled on the same computing node run in sequence. In reality, however, the tasks may be run in the time sharing manner. Our studies show that the discrepancy between the assumption of sequential execution and the reality of time sharing execution may lead to inaccurate calculation of the task makespan.

In this chapter, we first investigate the impact of the time sharing execution on the task makespan, and propose the method to model and determine the makespan with the time-sharing execution. Then we extend our work to a more complex and practical scenario: scheduling DAG applications for time sharing systems. Based on our time-sharing makespan model, we further develop the scheduling strategies for DAG jobs running in time-sharing. Extensive experiments have been conducted to verify the effectiveness of the proposed methods. The experimental results show that by taking time sharing into account, our DAG scheduling strategy can reduce the makespan significantly, comparing with its counterpart in sequential execution.

4.1 A Motivating Example

In this section, we present a case study to illustrate the difference of the time-sharing execution from the sequential execution and its impact on the DAG makespan. This case study considers a DAG job consisting of 7 tasks, whose

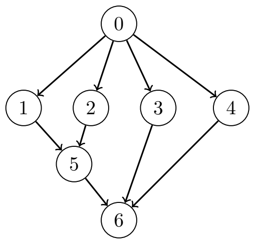


Figure 4.1: A Motivating Example DAG

topology is shown in Figure 4.1. The execution times of 7 tasks, t_0 to t_6, are $150, 200, 150, 50, 100, 100, 100$, respectively. There is no communication between tasks. Assume a scheduling decision of such a DAG on a set of two identical PMs (PM_1 and PM_2) is as follows. Tasks t_0, t_1, t_5, t_6 are scheduled to run in PM_1 while t_2, t_3, t_4 are in PM_2. If the tasks allocated to the same PM are run in sequence. Such a schedule leads to the minimal makespan. The corresponding critical path of the DAG is $t_0 \rightarrow t_1 \rightarrow t_5 \rightarrow t_6$.

The left figure in Figure 4.2 shows the sequential execution of the tasks in the two PMs. As shown in the figure, t_3 can only start the execution after task t_0 (which is its predecessor of t_3) and t_2 (which is scheduled to run before t_3) in PM_2 have finished. Other tasks have the similar execution precedence. With the sequential execution model, it is expected that t_5 starts the execution at the time point $350ms$, and the makespan of the DAG is $550ms$.

As discussed in the first section, when several tasks are allocated to the same PM, they will be run in the time-sharing manner by the OS. The right figure in Figure 4.2 shows the times-sharing execution of the tasks. As shown in the figure, t_2, t_3 and t_4 in PM_2 start execution concurrently after their predecessor t_0 finishes. t_2's finish time is then $450ms$, which is later than its finish time under the sequential execution model ($300ms$). This difference leads to the delay of t_5's start. In the time-sharing execution, t_5 starts the execution at $450ms$ with the delay of $100ms$ compared with the sequential execution. Consequently the actual DAG makespan with the time-sharing execution model is $650ms$, which is longer than the one expected with the sequential execution

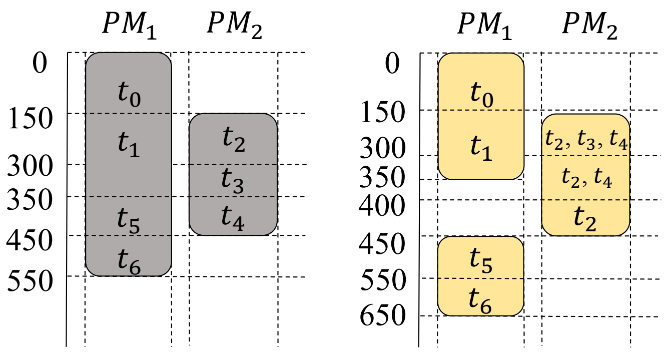


Figure 4.2: Sequential Execution Makespan (left) vs. Time-sharing Makespan (right)

($550ms$). Furthermore, the critical path in the time-sharing execution changes to $t_0 \rightarrow t_2 \rightarrow t_5 \rightarrow t_6$.

In most DAG scheduling algorithm in the literature, the scheduling decision is made based on the tasks' finish times, which are typically calculated by assuming the sequential execution. Although it is fine with the task scheduling in clusters, in which there is a centralized task queue in the head node and a task is sent to a computing node when the existing tasks running in the node have been completed. As discussed in the introduction, however, the tasks are run concurrently in distributed systems or virtualized systems. This may cause the discrepancy between the tasks' actual finish times and the finish times assumed by the task scheduler, as illustrated in this case study.

4.2 Workload and Resource Model

This section introduces the main notations used for the workloads and resources in this paper. A DAG-based application T is modelled as a directed acyclic graph (DAG) $G(V, E)$, where each task $t_i \in T$ is represented as a node $v_i \in V$. An edge e_{ij} from v_i to v_j, which is also denoted by (t_i, t_j), represents that there is the precedence constraint between tasks t_i and t_j. The weight of an edge represents the communication time TT_{ij} for sending the data from t_i to

t_j. Further, task t_i is called the predecessor of t_j, while t_j is the successor of t_i.

For task $t_i \in T$, its set of predecessors and successors, denoted by $pred(t_i)$ and $succ(t_i)$ respectively, are defined below:

$$pred(t_i) = \{t_j | t_j \in T \wedge (t_j, t_i) \in E\} \tag{4.1}$$

$$succ(t_i) = \{t_j | t_j \in T \wedge (t_i, t_j) \in E\} \tag{4.2}$$

Tasks without the predecessor or the successor are called the entry task or exit task, respectively.

In a DAG, the distance of a path is the sum of the execution times of all tasks and the weights of the edges (communication times) on the path. The critical path of a DAG is denoted as L. The makespan of a DAG is the distance of the critical path from the entry task to the exit task.

A cluster consists of a set of physical machines (PM), denoted by M, where $M = \{p_1, p_2, \ldots, p_s\}$. c_i denotes the processing capacity of p_i.

A task t_i is modelled by a tuple $t_i = \{st_i, ft_i, s_i, re_i\}$, where st_i is the time when t_i is *ready to start*(a task is ready to start only when all of its predecessors are completed and the relevant data sent by predecessors have been received by t_i); ft_i is the time when t_i is completed, which includes both the task's execution time and its data communication time; s_i is the size of the work (e.g., the number of instructions or the number of CPU cycles) that is to be performed in t_i; re_i is the current remaining work of t_i, which is calculated by the total work minus the finished work so far.

A Schedule is defined by $S = (G, M, Mapping)$, where G is the DAG graph, M is the cluster, and $Mapping$ is the mapping of the tasks in G to M. Figure 4.2 shows a exemplar schedule for scheduling a graph in figure 4.1 to a cluster of two PMs. In this example, $M = \{p_1, p_2\}$, $Mapping = \{1 : [t_0, t_1, t_5, t_6], 2 : [t_2, t_3, t_4]\}$.

After task t_i finishes the execution, it needs to send the results to its successors. We assume that the communication time can be neglected if the predecessor and the successor are mapped on the same PM. k_i is the number of successors of t_i. The total communication time of task t_i, denoted by TT_i, can be calculated by equation 4.3:

$$TT_i = \sum_{1}^{k_i} (TT_{ij} * l_{ij}) \tag{4.3}$$

$$l_{ij} = \begin{cases} 0 & \text{if } t_i \text{ and } t_j \text{ are on the same PM} \\ 1 & \text{otherwise} \end{cases} \tag{4.4}$$

Given the above workload and resource model, our objective is to investigate the impact of the time-sharing execution on the DAG makespan, and further propose the scheduling algorithms to mitigate the impact.

4.3 Makespan Model

4.3.1 The Makespan with the Sequential Execution Model

In sequential execution makespan model, tasks are regarded as executing in a one-by-one manner in a PM instance. At least they didn't take the time-sharing executing into consider when calculate finish time of the tasks. Thus, within a PM instance run queue, a ready task (i.e. that has received all results from its predecessors) can not start to execute before its previous task finishes.

Given a Schedule S, the start time st_i for task t_i can be determined by equation 4.6:

$$st_i = max\{lpft_i, prevft_i\} \tag{4.5}$$

where $lpft_i$ denotes the latest finish time of all t_i's predecessors, $prevft_i$ denotes the finish time of the task scheduled to run right before t_i.

The finish time ft_i for task t_i executed on PM_r can be derived by equation

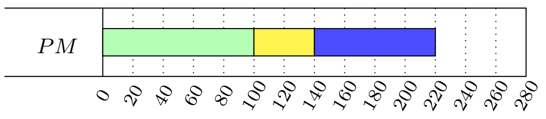


Figure 4.3: Sequential executing process

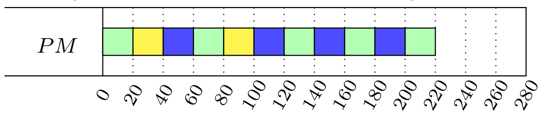


Figure 4.4: Time-sharing executing process

4.6:

$$ft_i = st_i + \frac{s_i}{PC_r} + TT_i \tag{4.6}$$

where s_i denotes the size of t_i, PC_r denotes the processing capacity of PM_r, TT_i denotes the total transferring time calculated by Equation 4.3.

Given the sequential execution, the makespan of a DAG can be calculated by applying Equation 4.5 and 4.6 iteratively from the entry task to the exit task in the DAG. However, actually, it is inaccurate. In fact, instead of executing one by one, tasks are executed in a time-sharing manner (i.e. round robin) within a PM instance. Round robin scheduler is widely used in current operating systems. Giving each job a time slot, make the jobs take turns to be executed. This is the scheduling principle of the most operating systems. Hence, the start time of the tasks will be different from sequential execution manner because tasks don't need to wait for the previous task finished. The finish time of the tasks will also be different because they may execute along with other tasks in a round robin manner. For example as shown in Figure 4.3 and Figure 4.4, we assume task t_i needs to execute 100 ms to complete. When it needs to take turns to run with other two tasks (the execution time of these tasks are 40 ms and 80 ms, respectively) in a round robin manner, it will need 220 ms to complete. Note that the scheduling time slot in this sample is 20 ms. In this 220 ms duration, task t_i will take 100 ms to execute and 120 ms to wait for the time slots (shown in Figure 4.4).

4.3.2 The Makespan model for Bag of Tasks under the Time-sharing Execution

Before presenting our makespan model for DAG applications, we first introduce a simple scenario: the makespan model for Bag of Tasks (BoT) under the time-sharing execution on a single core. Bag of Tasks is a set of independent tasks, denoted by $T = \{t_1, t_2, ..., t_n\}$. Suppose T runs on a Physical machine, PM_r, in a time-sharing manner. The execution time of each task $t_i \in T$ is determined by both the processing capacity PC_r of the PM and the number of tasks that are running concurrently with t_i (time-sharing execution). When calculating the finish time of t_i, we divide the task execution cycle of a task into a number of periods (the number of periods is denoted by m). A task is regarded as entering into a new execution period when the number of tasks running concurrently with t_i changes. For example in Figure 4.5, four tasks t_1-t_4 executing on the same PM in a time-sharing manner. The execution cycle can be divided into $m = 4$ time periods. In the first period, four tasks are sharing the processing capacity. When t_2 finishes, other tasks move into the second period, in which 3 tasks (t_1, t_3 and t_4) share the PM. The third and the fourth period are determined in the similar way. $time^s_j$ and $time^s_j$ represent the start and end time of the j-th period, respectively. $Share_j$ denotes the number of tasks that are running concurrently in the j-th period. Using the task size o_i and the time periods in the execution cycle, the execution time of task t_i can be derived by equation 4.7, where $(time^e_j - time^s_j) * PC_r * \frac{1}{Share_j}$ represents the useful work that t_i has completed in the j-th period.

$$o_i = \sum_{j=1}^{m} (time_{je} - time_{js}) * PC_r * \frac{1}{Share_j} \tag{4.7}$$

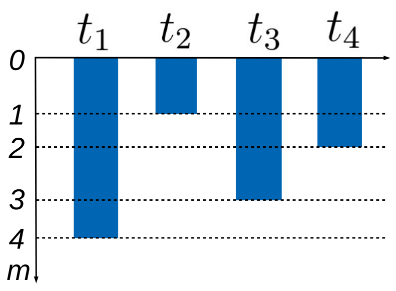


Figure 4.5: An example of four independent tasks executing on a single core

4.3.3 The Makespan Model for DAG Applications under the Time-sharing Execution

In this subsection, we present our method for computing the DAG makespan with the time-sharing execution model.

Given a Schedule S, the start time st_i of task t_i should be derived using Equation 4.8.

$$st_i = max\{lpft\} \tag{4.8}$$

Comparing Equation 4.8 with Equation 4.5 used for the sequential execution, the difference lies in the fact that task t_i does not have to wait for the completion of the tasks scheduled ahead of it. t_i can start once it is ready to run, i.e., all of its predecessors have finished.

Given a Schedule S, the finish time of task t_i is influenced by the processing capacity PC_r and the number of tasks that are running concurrently with t_i. When determining t_i's finish time, we divide the entire execution cycle of a task into a number of periods. t_i is regarded as moving into a new execution period when the number of tasks concurrently running with t_i changes.

For example in Figure 4.4, task t_0, t_1 and t_2 (represented by the color green, yellow and blue, respectively) are concurrently executing on a PM instance. We assume the size of t_0, t_1 and t_2 are 100, 40 and 80 respectively while the processing capacity PC of the PM is 1. According to equation 4.9, when calculating t_0's execution time, the first period is from $0ms$ to $120ms$. In this time period, the number of time-sharing tasks (i.e., $Share_1$) is 3. The useful work that t_0 completes in this time period is $(120 - 0) * 1/3 = 40$. At the time point $120ms$, t_1 finishes and the number of time-sharing tasks is reduced to 2 ($Share_2 = 2$), and therefore the execution moves into a new (second) period, which ends at the time point of $200ms$. In the second period, the useful work t_0 completes is $(200 - 120) * 1/2 = 40$. At the end of the second period, the remaining work of t_0 is 20. Similarly, when t_2 finishes, the execution enters into the final period in which there is only one task running ($Share_3 = 1$). In the final period, the useful work t_0 completes all its remaining work.

Assume the number of periods in the execution cycle of task t_i is m. m_j denotes the j-th period and $Share_j$ denotes the number of tasks that are concurrently running (time-sharing) with t_i. $time_{js}$ and $time_{je}$ denote the start and end time of period m_j, respectively. s_i denotes the size of t_i (e.g., the amount of work in terms of CPU cycles). Then equation 4.9 should hold, in which $(time_{je} - time_{js}) * \frac{PC_r}{Share_j}$ represents the amount of work completed (i.e., the number of CPU cycles dedicated to run t_i) during the period m_j.

$$s_i = \sum_{j=1}^{m}((time_{je} - time_{js}) * \frac{PC_r}{Share_j}) \tag{4.9}$$

Given s_i and a scheduling solution, we can determine at any time how many tasks are concurrently running with t_i. Consequently, we can determine m as well as the start and end time of each period (i.e., $time_{je}$ and $time_{js}$). With m, $time_{je}$ and $time_{js}$, we can determine the execution time of t_i, denoted by $time_{je}$ and $time_{js}$, using equation 4.10.

$$et_i = \sum_{j=1}^{m}(time_{je} - time_{js}) \tag{4.10}$$

The finish time of t_i can then be calculated by 4.11:

$$ft_i = et_i + TT_i \tag{4.11}$$

We apply equations 4.9, 4.10 and 4.11 iteratively for all tasks in a DAG from the entry task to the exit task. The finish time of the exit task is the makespan of the DAG.

Algorithm 4 is the pseudo-code of the algorithm to calculate the makespan of a DAG job when the tasks on the same machine run in the time-sharing manner. Given a Schedule $S = (G, PM, Mapping)$, the finish time of the tasks and the total makespan of the DAG application can be determined by this algorithm.

In the algorithm, the algorithm will estimate the remaining time of the running tasks on all PMs at the start of each time period, and identify the task with the minimum remaining time, which will be the task that finishes first among all time-sharing tasks. Also, when this shortest task finishes, its successors in the DAG, if any, can start running. The completion of this shortest task and the start of its successors may cause the change in the number of time-sharing tasks in the machine, and thus a new time period. If it is the case, the finish time of this shortest task will be the start time of the next time period. In addition, at the end of each time period, the algorithm will update the remaining work of the unfinished tasks on all machines.

Algorithm 4 Time-sharing Makespan Algorithm

```
Input: A Schedule strategy $S = (G, PM, Mapping)$, processing capacity $PC$,
       task size $t_i$
Output: Makespan of the Scheduled DAG application
29 Set $RunningList_r = []$ for each physical machine $PM_r$;
 $t_i = t_{entry}$;
 Add $t_{entry}$ to it's corresponding $RunningList$;
 $Start = 0$ ;
 while Not all tasks are finished do
30   From the $Start$ of this time period $m$:
       Update $RunningList$ for all $PM$s;
       Calculate the estimated $Remainingtime$ for all running tasks using:
       $TmpRemainingtime_i = re_i * Share_r / PC_r$ ;
       $duration = min\{TmpRemainingtime\} - Start$;
       $Start = min\{TmpRemainingtime\}$;
       Update remaining work $re$ for all running tasks using:
       $re_i = re_i - duration * PC_r / Share_r$;
       Check the finish task(s) in this period;
       Update these finish task(s) successor tasks' start time;
```

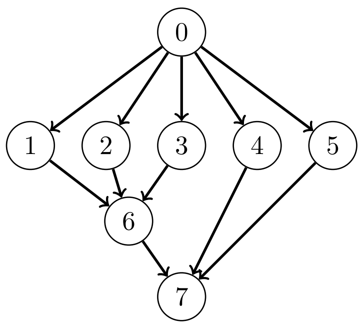


Figure 4.6: Case study: DAG

4.3.4 Case Study

We now present an example to show how our model derive the tasks' finish time and the makespan of the DAG under time-sharing execution. In addition, the process of sequential execution is also illustrated to show the difference.

Figure 5.3(e) shows the DAG of our case study. Tasks $\{t_0, ..., t_7\}$ are allocated on resource PM_0, PM_1 and PM_2. The mapping decision is as follows: $PM_0 = \{t_0, t_1, t_6, t_7\}$, $PM_1 = \{t_2, t_4\}$ and $PM_2 = \{t_3, t_5\}$. The task sizes are: $Size = \{t_0 : 150, t_1 : 300, t_2 : 150, t_3 : 250, t_4 : 150, t_5 : 100, t_6 : 200, t_7 : 100\}$. We assume that the the processing capacity PC of all physical machine instances is 1. The critical path of the DAG is $t_0 \rightarrow t_1 \rightarrow t_6 \rightarrow t_7$. The length of the critical path is 750. The working of the time-sharing makespan model is explained as follows. the calculation procedure is shown below. Note that $RunningList$ is abbreviated to RL, $TemporaryRemainingtime$ is abbreviated to $TmpRT$ in this section.

The entry task t_0 starts running at the start of the first period ($m = 1$), which is $0ms$. The Running List (RL) of three PMs are $RL_0 = \{t_0\}$, $RL_1 = \{\}$, $RL_2 = \{\}$. The Temporary (current) Remaining Time ($TmpRT$) of task t_0 is: $TmpRT_0 = s_0 * Share_0/PC = 150$. At time 150, t_0 finishes. t_0's successor tasks, t_1, t_2, t_3, t_4 and t_5, start. The execution enters into a new period ($m = 2$) since the number of sharing tasks changes.

In the second period starting from $150ms$, $RL_0 = \{t_1\}$, $RL_1 = \{t_2, t_4\}$, $RL_2 = \{t_3, t_5\}$; $TmpRT_1 = s_1 * Share_0/PC = 300$, $TmpRT_2 = 300$, $TmpRT_3 =$

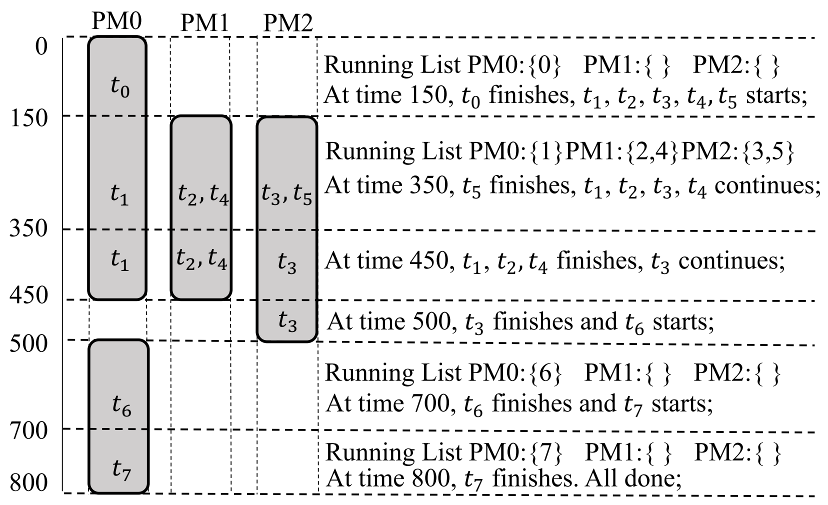

PM0
PM1
PM2
0
150
350
450
500
700
800
t_0
t_1
t_1
t_2, t_4
t_2, t_4
t_3, t_5
t_3
t_3
t_6
t_7
Running List PM0:{0} PM1:{ } PM2:{ }
At time 150, t_0 finishes, t_1, t_2, t_3, t_4, t_5 starts;
Running List PM0:{1}PM1:{2,4}PM2:{3,5}
At time 350, t_5 finishes, t_1, t_2, t_3, t_4 continues;
At time 450, t_1, t_2, t_4 finishes, t_3 continues;
At time 500, t_3 finishes and t_6 starts;
Running List PM0:{6} PM1:{ } PM2:{ }
At time 700, t_6 finishes and t_7 starts;
Running List PM0:{7} PM1:{ } PM2:{ }
At time 800, t_7 finishes. All done;

Figure 4.7: Case study: Scheduling scheme using Time-sharing Makespan model

500, $TmpRT_4 = 300$, $TmpRT_5 = 200$. t_5 is the task that has the minimum temporary remaining time among all tasks. The execution moves into a new period ($m = 3$) when t_5 finishes because the number of sharing tasks changes then. At the time point of 350, t_5 finishes and t_1, t_2, t_3 and t_4 continue to run. The remaining work of t_1 is: $TmpRT_1 = s_1 - duration * PC/Share_0 = 100$. Similarly, $Re_2 = 50$, $Re_3 = 150$ and $Re_4 = 50$.

In the third period ($m = 3$), starting from 350, $RL_0 = \{t_1\}$, $RL_1 = \{t_2, t_4\}$, $RL_2 = \{t_3\}$. t_1, t_2 and t_4 will be the first batch to finish at the same time (i.e., time 450) because they all have the same minimum temporary remaining time. After the third period ends, only t_3 continues. The remaining work of t_3 is $TmpRT_3 = TmpRT_3 - duration * PC/Share_2 = 50$.

In the final period ($m = 4$), starting from 450, t_3 executes in PM_2 alone, and finishes at the time point 500. Then t_6 starts. We can calculate the rest of the execution in the same way.

The sequential execution is illustrated in figure 4.8, in which the tasks allocated to the same PM are run one by one, which is not the reality. For example, the operating system in PM_1 will put task t_4 into execution without waiting for t_2 to complete, which leads to the inaccurate assumption of t_2's finish time,

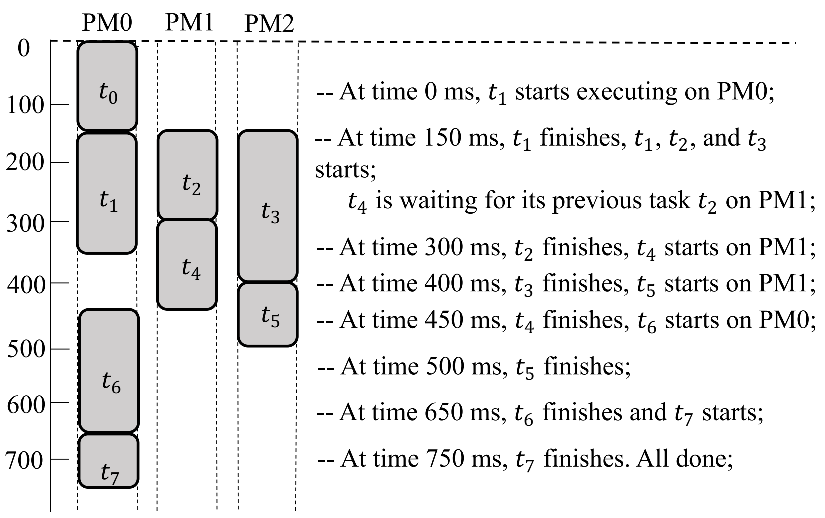

PM0
PM1
PM2
0
100
200
300
400
500
600
700
t_0
t_1
t_2
t_3
t_4
t_5
t_6
t_7
-- At time 0 ms, t_1 starts executing on PM0;
-- At time 150 ms, t_1 finishes, t_1, t_2, and t_3 starts;
t_4 is waiting for its previous task t_2 on PM1;
-- At time 300 ms, t_2 finishes, t_4 starts on PM1;
-- At time 400 ms, t_3 finishes, t_5 starts on PM1;
-- At time 450 ms, t_4 finishes, t_6 starts on PM0;
-- At time 500 ms, t_5 finishes;
-- At time 650 ms, t_6 finishes and t_7 starts;
-- At time 750 ms, t_7 finishes. All done;

Figure 4.8: Case study: Scheduling scheme using Traditional Makespan model

and may consequently affect its successors' start time.

4.4 DAG Scheduling Adjustment

Given a DAG $G = (V, E)$ and its schedule S, we can calculate the makespans under the time-sharing execution (called time-sharing makespan) and under the sequential execution (called sequential makespan). With the same schedule S, we find that in theory the time-sharing makespan is always longer than the sequential makespan. This is because the time-sharing execution causes some tasks to have later finish times than those under the sequential execution. These later finish times may in turn delay the start of their successors and eventually result in a longer makspan. In other words, the actual makespan of a DAG is always longer than the makespan assumed under the sequential execution.

Since the assumed makespan is always shorter than the actual makespan, we regard the assumed makespan as a deadline *ddl*. We try to use the *ddl* as the target and reduce the actual makespan by adjusting the schedule S.

In order to determine whether the deadline *ddl* is met by a particular schedule, we first calculate the latest finish time of the tasks in a DAG, which is presented in subsection 4.4.1. The adjustment process is performed by migrating the tasks which cannot finish by their latest finish time to another suitable PM (only using the current PMs in the schedule S without adding new PMs). In order to make adjustment decisions, we need to evaluate the impact of a particular migration on the existing tasks running on the PM which is the migration destination (called destination PM). The evaluation of migration impact is presented in subsection 4.4.2. Finally, the schedule adjustment algorithm is presented in subsection 4.4.3.

4.4.1 Calculating Latest Finish Time

When mapping tasks to PM instances, firstly, we need to consider the order of the tasks. In our strategy, the makespan of the given DAG schedule S by assuming the sequential execution, which we call makespan in sequential execution, is used as the deadline *ddl* for our schedule adjustment. We then use the following equation to derive the latest start time lst of every task in order to meet the deadline. Using the *ddl*, we can derive tasks' latest start time lst by their decreasing topological level:

$$lst_i = \begin{cases} ddl - \frac{s_i}{PC_r} & \text{if } t_i = t_{exit} \\ \min\limits_{t_s \in succ(t_i)} \{lst_s - \frac{s_i}{PC_r}\} & \text{otherwise} \end{cases} \tag{4.12}$$

Latest start time lst value indicates the allocation urgency of the DAG. Earlier lst_i earns higher priority. Thus, we can get a $OrderList$ sorted by increasing lst and allocate the tasks from the highest priority task (i.e. t_{entry}) to the lowest priority task (i.e. t_{exit}).

Similarly, we can use equation 4.13 to derive the latest finish time lft of every task in the DAG. Every task, t_i should finish by its latest finish time lft_i. Otherwise, the DAG will not meet the deadline.

$$lft_i = \begin{cases} ddl & \text{if } t_i = t_{exit} \\ \min\limits_{t_m \in succ(t_i)} \{lft_m - \frac{s_m}{PC_r}\} & \text{otherwise} \end{cases} \tag{4.13}$$

lft_i is used to determine the tasks whose allocations need to be adjusted. With equation 4.11, we can calculate the actual finish time ft_i of every task. If ft_i is greater than lft_i calculated by equation 4.13. The allocation of Task t_i needs to be adjusted, which is stored in an *AdjustList* in the increasing order of the task's latest start time (lst_i). For each task t_i in *AdjustList*, we try to migrate it to another PM so that ft_i can be no more than lft_i. We deem the adjustment of the Schedule S to be successful only when all tasks in *AdjustList* can find their suitable PMs. The task migration algorithm will be introduced in detail later.

4.4.2 Calculating the Migration Impact

In this subsection, we first present some simple migration cases (in subsubsection 4.4.2) to show how the executions of the existing tasks in the destination PM are affected by the migration. Then we present the impact of task migration in more complicated cases.

Simple migration cases

Assume Task t_k starts at time st_k and finishes at ft_k in a PM. When we migrate t_k to another PM (e.g., PM_x), the finish times of the tasks that are allocated to run on this destination PM PM_x may be affected. In the simple migration cases, we assume that there is only one task (e.g., t_i) in PM_x. We find that the alignment relation between t_i and t_k (i.e., the comparison between the start/end times of t_k and t_i) determines how t_i will be affected by the migration of t_k.

In Figure 4.9, we draw four cases (t_1-t_4 in PM_a-PM_d) which have different alignment relation with t_k.

a) In PM_a, $st_1 \leq st_k$ and $ft_1 \leq ft_k$. In this case, the new finish time of t_1 after the migration will be:

$$ft'_1 = st_k + (ft_1 - st_k) * 2 \tag{4.14}$$

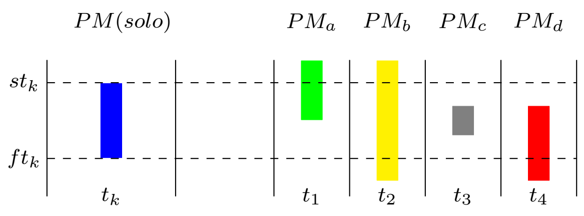


Figure 4.9: Migration effects on four kinds of tasks

b) In PM_b, $st_2 \leq st_k$ and $ft_2 > ft_k$. In this case, the new finish time of t_2 will become:

$$ft'_2 = ft_2 + (ft_k - st_k) \tag{4.15}$$

c) In PM_c, $st_3 > st_k$ and $ft_3 \leq ft_k$. In this case, the new finish time of t_3 will become:

$$ft'_3 = st_3 + (ft_3 - st_3) * 2 \tag{4.16}$$

d) In PM_d, $st_4 > st_k$ and $ft_4 > ft_k$. In this case, the new finish time of t_4 will become:

$$ft'_4 = \begin{cases} st_4 + (ft_4 - st_4) * 2 & \text{if } ft'_4 < ft'_k \\ ft_4 + (ft_k - st_4) & \text{if } ft'_4 > ft'_k \end{cases} \tag{4.17}$$

More complicated migration cases

In the destination PM, there may exist more than one tasks that are affected by migration. For example, in the time-sharing environment, the four tasks t_1-t_4 in figure 4.9 are allocated to the same PM. In this case, calculating the migration impact becomes more complex. In addition, the affected tasks may further affect other co-running tasks and/or delay the start of their successor tasks. In this subsection, we propose an algorithm to derive the affected finish times of the affected tasks in this complicated cases.

Given a Schedule S, we can get the start time st_i and finish time ft_i of the tasks within the DAG; Assume task t_k in the schedule is migrated to another PM, the resulting schedule is denoted by S'. We can also calculate the we can know the migrated task (say t_k) as well as its start time st_k.

The pseudo-code of deriving the finish time ft'_i of the tasks affected by the migration is shown in Algorithm 5. Only when the finish time of a task (ft_i) in the destination PM is later than the start time of t_k (st_k) (Index 31 in the algorithm), will the task will be affected by the migration of t_k. In the algorithm, Index 31-32 find all tasks that may be affected by the migration, initialize their new start and finish times, create a dictionary *Re_cal* to store the potentially affected tasks as well as their existing start times st_i, finish times ft_i, new start times st'_i and new finish times ft'_i. The naive method of examining the impact of the migration is to calculate the finish times of all potentially affected tasks (i.e., the tasks in *Re_cal*) and check with the new finish times change. We propose a better method as follows to reduce the calculations needed. The algorithm calculates the topology level of those potentially affected tasks in the DAG and creates a dictionary *l_dict* to store their DAG topology levels. The tasks' topology levels are used as the information to terminate the while loop indexed by 34 in the algorithm. The termination condition of the while loop is that: when the new finish times of all tasks ft'_i in a level (e.g., level l_i) are the same as their existing finish times ft_i, the migration effect stops at this topology level and will not propagate further to lower topology levels. In other word, the successor tasks of this level will not be affected by the migration. Index 33 identifies the first task in *Re_cal* (i.e. with minimum start time) and its topology level l_{min}. The calculation in the while loop starts from this task. The code segment in index 6 calculates the start and finish times of the tasks that may be affected by the migration, until the termination condition discussed above is met.

4.4.3 Task Migration Algorithm

The condition for a successful task migration is that in the destination PM there are no tasks (including the task to be migrated and the existing tasks in the PM) that will miss their latest finish time.

In this section, we present a task migration algorithm to adjust the given

Algorithm 5 Single-task Migration Effect Calculation

```
Input: A Schedule strategy $S = (G, PM, Mapping)$, the schedule strategy $S'$
       after migration, processing capacity $PC$, task size $s_i$
Output: Finish time $ft'_i$ of the affected tasks
31 for each $ft_i > st_k$ do
32     add task $t_i$ to the $Re\_cal$ dictionary;
         $st'_i, ft'_i = 0$;
         $Re\_cal[t_i] = [st_i, ft_i, st'_i, ft'_i]$;
         Calculate $t_i$'s topology level $l_i$ in $G$;
         $l\_dict[l_i].append(t_i)$;
33 $Start = \min_{Re\_cal}\{st_i\}$ marked as task $t_{min}$;
   $t_i = t_{min}$;
   while Not all $ft'_i = ft_i$ in level $l_i$ of $S'$ do
34     From the $Start$ of this time period $m$:
         Update $RunningList$ for all $PM$s;
         Calculate the estimated $Remainingtime$ for all running tasks using:
         $TmpRemainingtime_i = re_i * Share_r / PC_r$ ;
         $duration = min\{TmpRemainingtime\} - Start$;
         $Start = min\{TmpRemainingtime\}$;
         Update remaining work $re$ for all running tasks using:
         $re_i = re_i - duration * PC_r / Share_r$;
         Check the finish task(s) in this period;
         Update these finish task(s) successor tasks' start time $st'_i$;
         $l_i = l\_dict.keys()[l\_dict.values().index(t_i)]$;
35 for all unaffected tasks in $Re\_cal$ do
36     $ft'_i = ft_i$;
```

schedule and reduce the actual makespan towards the set deadline. First, the algorithm recognizes the tasks that need to be migrated in a PM (i.e., those tasks that miss their latest finish time according to the set deadline), and define the *AdjustList* to store the tasks in the increasing order of their latest start times (i.e. urgency). Then the algorithm tries to migrate these tasks one at a time. For a task (e.g., t_k) to be migrated, the algorithm tries to find a suitable PM in which t_k is able to catch its latest finish time lft_k and all existing tasks in the PM do not violate their latest finish time (determined by using Algorithm 5) due to this migration. Only when all tasks in *AdjustList* can find their suitable PMs, we regard Schedule S as being adjustable. The output of this algorithm is whether S is adjustable (0 or 1), the new S' if it is adjustable and its corresponding real makespan under the time sharing execution.

The pseudo-code is shown in Algorithm 6. Index 37-39 is the preprocessing phase. In this phase, the algorithm calculates all tasks' slack time, topology level and identify all the tasks that need to be migrated (marked as t_k). Index 44-51 tries to find a suitable PM for the current task to be migrated. The algorithm introduces a metric, $min\_Slack_{l_k}^{PM_r}$, which refers to the minimum slack time of the tasks that are in the same topology level as t_k within PM_r. $S\_Slack_{l_k}^{PM_r}$ is calculated using Equation 4.18.

$$min\_Slack_{l_k}^{PM_r} = \min_{\{t_k \in Affec\_list\}} (lft_k - ft_k) \tag{4.18}$$

When looking for a suitable PM for t_k, we calculate $S\_Slack_{l_k}^{PM_r}$ for every PM except the source machine (the machine which t_k currently resides in). Bigger value of $min\_Slack_{l_k}^{PM_r}$ indicates a PM with higher tolerance to accept the migration of t_k. The algorithm defines a PM_List that sorts the PMs in the decreasing order of $min\_Slack_{l_k}^{PM_r}$ (Index 44). The algorithm checks from the first PM in PM_List whether t_k can be migrated to this PM (labelled as PM_{try}). The checking is performed in the following way. The algorithm calculates the new finish times of the affected tasks by using Algorithm 5. If

Algorithm 6 Task Migration Algorithm

```
Input: DAG $G$ and Schedule $S = (G, PM, Mapping)$, processing capacity $PC$,
       task size $s$
Output: Whether $S$ is adjustable (0 or 1), Adjusted $S$' and its corresponding
        real makespan
37 for All tasks in DAG $G$ do
38     Calculate $t_i$'s real finish time $ft_i$ using time-sharing makespan model;
       Calculate $t_i$'s latest finish time $lft_i$ using eq. 4.13 ;
       $slack_i = lft_i - ft_i$;
       Calculate $t_i$'s topology level $l_i$ in $G$;
       $l\_dict[l_i].append(t_i)$;
       if $slack_i < 0$ then
39         Add $t_i$ to $AdjustList$;
40 Sort $AdjustList$ by increasing $lst$ derived from eq. 4.12;
   for tasks (marked as $t_k$) in ordered $AdjustList$ do
41     Mark $t_k$'s current allocated $PM$ as $PM_{cur}$ ;
       for All $PMs$ (marked as $PM_r$) except $PM_{cur}$ do
42         Add $PM_r$ to $PM\_List$;
           $min\_Slack_{l_k}^{PM_r} = 0$;
           for task $t_a \in PM_r$ and $l_a = l_k$ do
43             $min\_Slack_{l_k}^{PM_r} = min\{slack_a\}$;
44     Sort $PM\_List$ by decreasing $min\_Slack_l^{PM}$;
       $PM_{try} = PM\_List[0]$;
       while $PM\_List$ is not empty do
45         Assume $t_k$ changes its allocation to $PM_{try}$;
           Make $Affec\_list$ of the Affected tasks and calculate their $ft'$ calling
           Algorithm 5;
           for each task (marked as $t_{aff}$) in $Affec\_list$ do
46             $slack'_{aff} = ft'_{aff} - lft_{aff}$;
               if $slack'_{aff} < 0$ then
47                 $PM_{try}$ is not a suitable $PM$ to migrate;
                   Remove $PM_{try}$ from $PM\_List$;
                   Break
48         if No more $lft$ missing happens then
49             Migrate $t_k$ to $PM_{try}$;
               Update all corresponding information;
               Break;
50     if There is no $PM$ changeable for $t_k$ then
51         Schedule $S$ is non-adjustable;
           Exit;
52 if Schedule $S$ is adjustable then
53     Update the adjusted Schedule $S'$;
       Calculate the corresponding real makespan;
```

the new finish time of any affected task is bigger than the latest finish time of the task (i.e., the new slack time of the affected task is less than 0), then the PM is not a suitable PM and the algorithm moves on to check next PM. Otherwise, t_k is migrated to this PM (Index 48-49). If the algorithm cannot find a suitable PM for t_k, this schedule S is regarded as non-adjustable (Index 50-51). In other words, the algorithm cannot reduce the time-sharing makespan to the sequential makespan by only adjusting the schedule.

4.5 Multi-task Migration Algorithm

In last section, the algorithm migrates one task at a time. We find that such a migration strategy is inefficient and can cause unnecessary calculations. In this section, we present an improved strategy for the scheduling adjustment. In the improved strategy, multiple tasks are considered together for migration, which we call multi-task migration. More specifically, the algorithm tries to migrate at a time all tasks in the same topology level that miss their latest finish times. The pseudo-code is presented in Algorithm 7.

We now give an example to show how the algorithm works. For a DAG and its corresponding Schedule S, the set of tasks to be adjusted (denoted by $Adjust\_dict$), and the set of tasks that do not need to be adjusted (denoted by non_Adjust_dict) are as follows.

$Adjust\_dict = \{1 : [t_3, t_5, t_7], 2 : [t_{14}, t_{16}], ...\}$

$non\_Adjust\_dict = \{0 : [t_0], 1 : [t_2, t_4, t_6, t_8, t_9], ...\}$

Firstly, the algorithm temporarily remove these three tasks from S, as shown in figure 4.11, and then calculates the temporary finish time tmp_ft of the rest non-adjusted tasks as well as the temporary total slack time $tmp\_S\_Slack_{l_i}^{PM_r}$ of all non-adjusted tasks in each PM. Then the algorithm tries to re-allocate the three tasks back to the PMs. The three tasks are ordered by the decreasing execution time. The tasks are re-allocated in the order of the largest execution time first. A task is re-allocated to the PM that has the largest $tmp\_S\_Slack_{l_i}^{PM_r}$.

Algorithm 7 Multi-Task Migration Strategy

Input: DAG G and Schedule $S = (G, PM, Mapping)$, processing capacity PC, task size s

Output: Whether S is adjustable (0 or 1), Adjusted S' and its corresponding real makespan

```
for All tasks in DAG G do
    Preprocessing calling relative functions in Algorithm 6;
    if $slack_i < 0$ then
        Add $t_i$ to $AdjustList$;
        $Adjust\_dict[l_i].append(t_i)$;
    if $slack_i \geq 0$ then
        Add $t_i$ to $non\_AdjustList$;
        $non\_Adjust\_dict[l_i].append(t_i)$;
for $l_i$ in Adjust_dict ordered by increasing level do
    Set $Number$ to be the task number in $l_i$;
    while Not all tasks in $l_i$ satisfy slack $\geq 0$ and $Number > 0$ do
        Assume that tasks in $Adjust\_dict[l_i]$ are removed from $S$;
        for all tasks in $non\_Adjust\_dict[l_i]$ do
            Calculate the $tmp\_ft$ and $tmp\_slack$;
        Derive $tmp\_S\_Slack_{l_i}^{PM_r}$ for each $PM$;
        Sort $PM\_List$ by decreasing $tmp\_S\_Slack_l^{PM}$;
        Re-map tasks in $Adjust\_dict[l_i]$ using $LTF$ to ordered $PM\_List$;
        for all tasks in level $l_i$ do
            Check task's new slack time;
            if $\exists t_{exist} \in l_i, slack'_{exist} < 0$ then
                Add these tasks to $Adjust\_dict[l_i]$;
        $Number = Number - 1$;
    if $Number == 0$ then
        Schedule $S$ is non-adjustable;
        exit;
if Schedule S is adjustable then
    Update the adjusted Schedule $S'$;
    Calculate the corresponding real makespan;
```

For example, in this case, the order of the execution time is $t_5 > t_3 > t_7$. The algorithm first allocates t_5 to the PM with the maximum temporary total slack time (i.e. PM_b), then deducts t_5' execution time from PM_b's temporary slack time and re-orders the PMs. Next, t_3 is re-allocated in the same way. After we finish re-allocating all tasks in level 1, we re-calculate all tasks' finish times in this level and check if any task misses its latest finish time. The algorithm only moves on to adjust the tasks in the next topology level after there is no missing of latest finish time in the current level. If there exists the tasks in the current level which miss their latest finish time, the algorithm adds these tasks to *Adjust_dict* and continue to adjust the current level until there is no task that misses its latest finish time. Then, the algorithm calculates and updates the new finish time of the whole DAG. The algorithm sets a *number* (Index 38 in Algorithm 7) as the maximum number of times the algorithm re-allocate the batch of tasks in the current level and checks whether there is missing of latest finish time. The *number* is set to be the number of tasks in current level.

The differences between single-task migration strategy and multi-task migration strategy are: 1) Single-task migration strategy takes out one task at a time and migrates it, while multi-task migration strategy takes out all tasks in a topology level at a time and migrates the batch of tasks at a time; 2) Single-task migration strategy does not allow any affected task to miss its latest finish time during the migration of a task, while in multi-task migration, the algorithm only checks whether there is missing of latest finish time after all tasks in the batch have been allocated. The multi-task migration algorithm does not allow any missing of latest finish time in the current level, but allows the successor tasks (i.e., the tasks in the lower topology levels) to miss their latest finish time. If there are successor tasks that miss the latest finish time, they are added to *Adjust_dict* and will be considered for migration later.

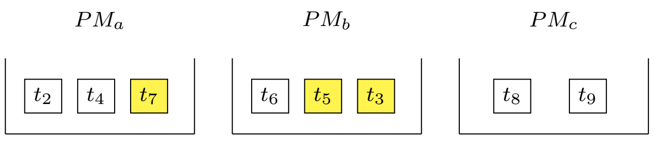


Figure 4.10: Original Mapping for topology level 1

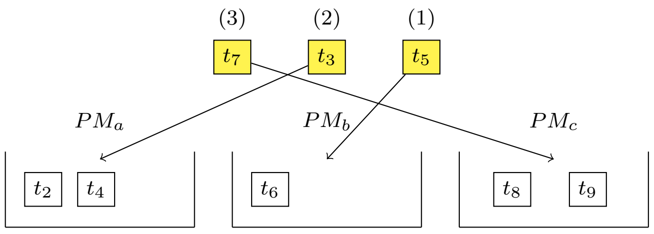


Figure 4.11: Re-allocation for topology level 1

4.6 Task Allocation Algorithm

Not all task schedules can be adjusted to meet the deadline. If the task migration algorithm fail to reach a successful adjustment. We develop a Task Allocation Algorithm (TAA) to find a task schedule from scratch for the time-sharing execution. TAA assumes the same number of PMs as that in the schedule S generated for sequential execution.

In TAA, we still use the makespan in sequential execution as the deadline (target) for finding the schedule solution in time-sharing. TAA generates an *Orderlist* in the similar way as we construct the *Ajustlist* in Task Migration Strategy. For each task in *Orderlist*, TAA tries to allocate it to a best PM based on a metric we propose, which is called Total deadline Miss Time (tmt). tmt is defined as the total of all deadline misses in a PM. The pseudo-code of TAA is shown in Algorithm 8. In Index 71, sequential execution makespan of schedule S is calculated and set to be the deadline. Index 72 calculates the latest start time lst, latest finish time lft of all tasks and makes a $OrderList$, in which the tasks are sorted by the increasing topological level. Within the same topological level, tasks are sorted by increasing latest start time lst. Then we allocate the tasks from the front to the end of the $OrderList$. When we need to make a allocative decision for task t_i, we first temporary allocate the task to all PMs and find a best PM for it. Assuming that allocate task t_i to PM_s, we calculate all allocated tasks' temporary finish time by our time-sharing model based on this temporary allocation. Using these temporary finish times and tasks' lft, we can derive the total latest finish time missing tmt of each PM and choose the PM that has minimum tmt to allocate t_i. If there are more than one PM have no lft missing (i.e. $tmt = 0$), we will choose the PM that has maximum total slack time tst to allocate t_i. After making an allocative decision for task t_i, Index 83 updates the schedule S', as well as the corresponding tasks' finish times and their child tasks' start times.

Algorithm 8 Task Allocation Algorithm

```
Input: DAG $G$ and Schedule $S = (G, PM, Mapping)$
Output: A newly Schedule $S'$ and its Makespan
71 Calculate $S$'s sequential execution makespan and set to $ddl$;
   Calculate the $G$'s topological level $level$;
   for All tasks in DAG $G$ do
72     Calculate $t_i$'s $lst_i$ and $lft_i$;
73 for level from 0 to the highest topological level do
74     Sort the tasks on the same $level$ by increasing $lst$ and add to the $OrderList$;
75 for From front to back of the OrderList do
76     for all $PMs$ do
77         Calculate $ft$ of all allocated tasks;
           Calculate the total $ddl$ missing time $tmt_s$;
           if total ddl missing time $tmt_s = 0$ then
78             Calculate the total $ddl$ slack time $tst_s$;
79     if there are >1 proposed $S'$ has total $tmt_s = 0$ then
80         Allocate $t_i$ to the $PM$ with $max(tst_s)$;
81     else
82         Allocate $t_i$ to the $PM$ with $min(tmt_s)$;
83     Update $S'$, all related tasks' $ft$ and child tasks' $st$;
```

4.6.1 Resource Bounds

In this subsection, based on the features of DAG deadline constrained application, we determine the upper bound for the minimum number of PM instances that need to guarantee the DAG deadline.

Lemma 1. *Given a deadline constrained DAG application $G = (V, E)$, the critical path of the DAG be CP, critical path execution time be TCP, deadline be ddl ($ddl > TCP$), and the number of PM instances needed to guarantee the applications deadline be M. M is equal to the fattest level task number. Then we have: there exists at least one Schedule S that can meet the application's deadline.*

Proof. The number of PM instances is equal to the fattest level task numbers. Thus, it is possible to allocate all tasks in the same level to different PMs. When we guarantee that all same level tasks can be executed on different PMs, the makespan of the DAG will be equal to the critical path execution time TCP. And $TCP < ddl$. So it is obvious that there exists at least one Schedule to meet the application's deadline. $\square$

4.7 Evaluation

To facilitate the evaluation of the workflow algorithms, Pegasus has developed a set of synthetic workflow generators. These generators use the information gathered from actual executions of scientific workflows to generate realistic, synthetic workflows resembling those used by real world scientific applications. These workflows come from [38] are widely used in this research field. In this section, we use these real-world workflows for evaluation. In the experiments, we compare the Makespan in Sequential execution (denoted by makespan-S, which is the makespan by assuming the sequential execution), the Makespan in Time-sharing execution (denoted by makespan-TS, which is the makespan of the DAG when the tasks are run in time-sharing in reality) and the makespan obtained by TAA (denoted by makespan-TAA). Makespan-S and Makespan-TS are computed using the makespan models presented in Section 4.3.

In this section, we use both Real-World Workflow and randomly generated DAG for evaluation. The Real-World Workflow characteristics including task

Table 4.1: Characteristics of the Real-World DAGs

	Task Num.	**Edge Num.**	**Sequence Runtime(s)**
Montage 25	25	45	227.75
Montage 50	50	106	508.64
Montage 100	100	233	1079.34
Montage 1,000	1,000	2485	11378.69
Epigenomics 24	24	27	17720.15
Epigenomics 46	46	54	41401.78
Epigenomics 100	100	122	403400.2
Epigenomics 997	997	1234	3854768.81
CyberShake 30	30	52	828.65
CyberShake 50	50	92	1322.98
CyberShake 100	100	192	2236.61
CyberShake 1,000	1,000	1976	23319.19
Sipht 30	30	33	5546.45
Sipht 60	60	66	11668.91
Sipht 100	100	109	17379.73
Sipht 1,000	1,000	1096	173678.19
Inspiral 30	30	35	6617.07
Inspiral 50	50	60	11761.95
Inspiral 100	100	119	21023.96
Inspiral 1,000	1,000	1233	227702.63

number, edge number and sequence runtime are given in Table 4.1. Sequence runtime indicates the sum of all tasks' runtime in the DAG. The structures of different applications are given in Figure 4.17.

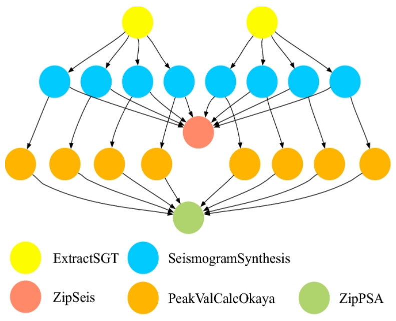


Figure 4.12: CyberShake

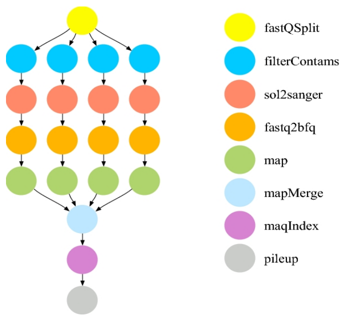


Figure 4.13: Epigenomics

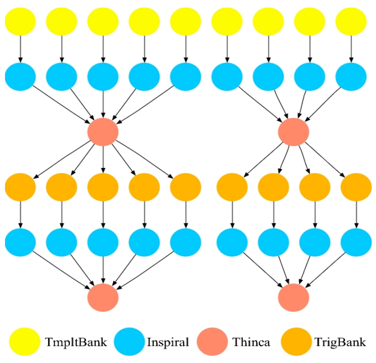


Figure 4.14: Inspiral

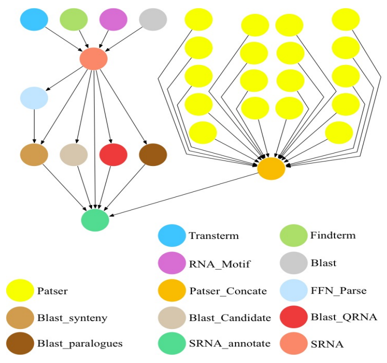


Figure 4.15: Sipht

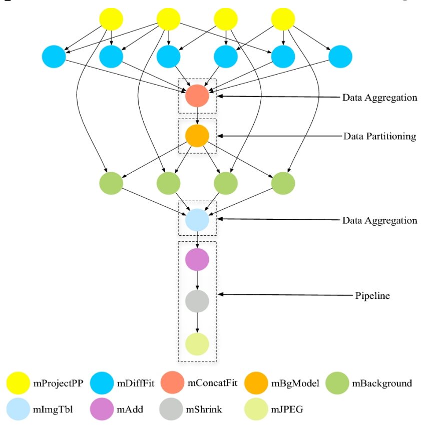


Figure 4.16: Montage

Figure 4.17: Structures of the real-world workflows [21]

4.7.1 Performance with different number of tasks

Figure 4.18 shows the performance of the real-world workflows with different number of tasks in terms of makespan-S, makespan-TS and makespan-TAS.

Montage

Montage has been created by the NASA/IPAC Infrared Science Archive that can be used to generate custom mosaics of the sky using input images in the Flexible Image Transport System (FITS) format. Figure 4.18(a) - 4.18(d) shows the gaps among makespan-S, makespan-TS and makespan-TAS. The results indicates that there indeed exits the gap among these makespans. Our TAS algorithm can reduce the realistic makespan by taking the time-sharing execution into account.

Epigenomics

This workflow is being used by the Epigenome Center in the processing of production DNA methylation and histone modification data. It has the largely pipelined tasks and a large degree of parallelism. For example, Epigenomics 997 has 7 entry tasks and a parallel degree of 250. Due to its DAG structure, there is not a big difference between makespan-S and makespan-TS. However, comparing with makespan-S, TAS improves the makespan by 8.88%, 7.9%, 10.7% and 14.3% with 24, 46, 100 and 997 tasks, respectively.

CyberShake

The Cybershake workflow is used by the Southern California Earthquake Center (SCEC) to characterize the earthquake hazards in a region using the Probabilistic Seismic Hazard Analysis (PSHA) technique. Figures 4.18(i) - 4.18(l) show a big difference between makespan-S and makespan-TS: 33.1%, 20.4%, 7.4% and 2.52% with 30, 50, 100 and 1,000 tasks, respectively. Given the limited number of PMs in the experiments (less than the parallel degree of the workflow), the DAG with the flat structure often cause a big difference between makespan-S and makespan-TS since the time-sharing execution results in the big delay in some tasks' finish time comparing with the sequential execution. TAS shows a outstanding optimization ability, improving the makespan by 43.56% and 41.82% with 100 nodes and 1,000 tasks, respectively.

Sipht

The Sipht workflow is used to automate the search for sRNA encoding-genes for all of the bacterial replicons in the National Center for Biotechnology Information (NCBI) database. It is a highly parallel, flat structured DAG application. Figures 4.18(n) - 4.18(p) show the gaps of 345.58s, 696.81s and 63s between makespan-S and makespan-TS with 60, 100 and 1,000 tasks, respectively. However there is no noticeable difference between two makespans when the number of tasks is less than 30 no matter how many PMs are used.

Inspiral

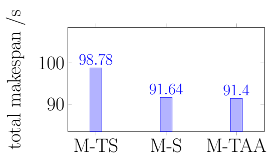
The LIGO Inspiral Analysis Workflow is used to analyze the data obtained from the coalescing of compact binary systems such as binary neutron stars and black holes. The parallel degree of the DAGs are 7, 12, 23 and 229 with 30, 50, 100 and 1,000 tasks, respectively. There is a gap of 1.87%, 1.61%, 25.2% and 25.7% between makespan-S and makespan-TS with 30, 50, 100 and 1,000 tasks respectively. TAS shows a makespan improvement of 17.6%, 1.18%, 0.01% and 14.7% with 30, 50, 100 and 1,000 tasks, respectively.

4.7.2 Performance with the different number of PMs

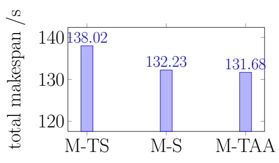
Table 4.2 and Figure 4.19 show the makespan of the real-world workflows with 50 and 100 tasks, respectively, when using different number of PMs. As can be seen from Table 4.2, different number of PMs lead to the different gaps between makespan-S and makespan-TS. The decrease of the makespan is not linear with the increase of the number of PM. When the number of PM reaches the degree of the parallelism of the DAG, the gap disappears. In our experiment, the parallel degrees are 15, 10, 23, 50 and 12 for Montage 50, Epigenomics 46, CyberShake 50, Sipht 60 and Inspiral 50 respectively in Table 4.2; the parallel degrees are 60, 24, 46, 89 and 23 for Montage 100, Epigenomics 100, CyberShake 100, Sipht 100 and Inspiral 100 respectively. For flat and highly parallel DAG such as "Sipht 100", varying the number of PMs (i.e. 12 - 28) makes almost no difference to the makespan when the number of PM is far less than the parallel degree (i.e. 89 in this case).

4.7.3 Results for Randomly Generated DAGs

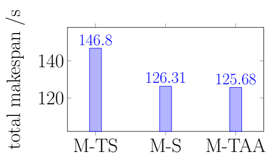
We randomly generate 10 DAGs, each of them is comprised of 30 tasks. These DAGs are allocated to 6 PMs and 8 PMs, respectively. Figure 4.20 shows



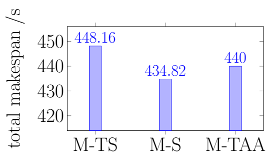
(a) Mon 25 ($PM = 3$)



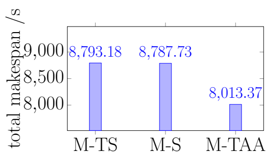
(b) Mon 50 ($PM = 5$)



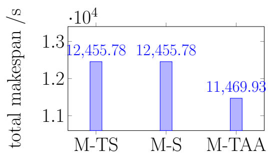
(c) Mon 100 ($PM = 14$)



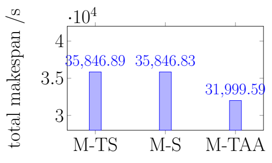
(d) Mon 1k ($PM = 140$)



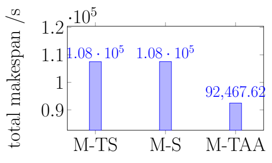
(e) Epi 24 ($PM = 3$)



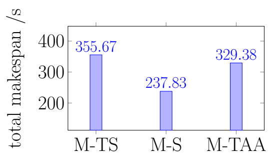
(f) Epi 46 ($PM = 5$)



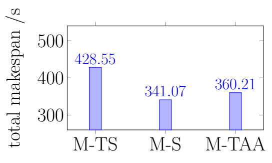
(g) Epi 100 ($PM = 22$)



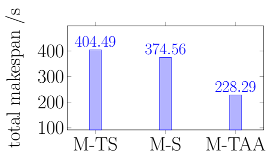
(h) Epi 997 ($PM = 50$)



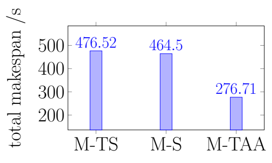
(i) Cyb 30 ($PM = 3$)



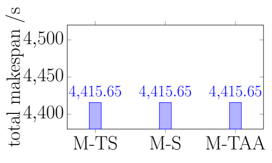
(j) Cyb 50 ($PM = 5$)



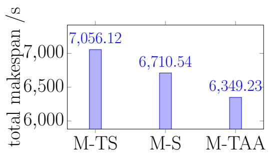
(k) Cyb 100 ($PM = 24$)



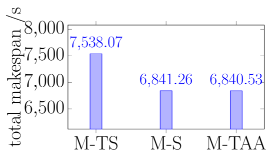
(l) Cyb 1k ($PM = 140$)



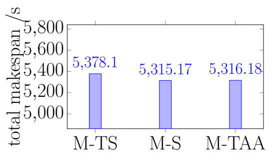
(m) Si 30 ($PM = 4$)



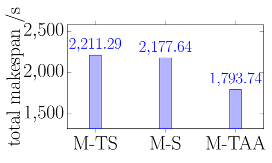
(n) Si 60 ($PM = 15$)



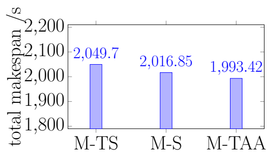
(o) Si 100 ($PM = 20$)



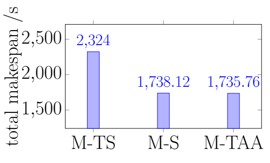
(p) Si 1k ($PM = 140$)



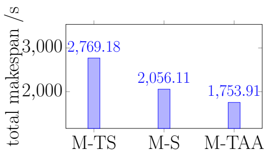
(q) Ins 30 ($PM = 5$)



(r) Ins 50 ($PM = 10$)



(s) Ins 100 ($PM = 20$)

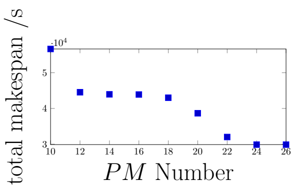


(t) Ins 1k ($PM = 200$)

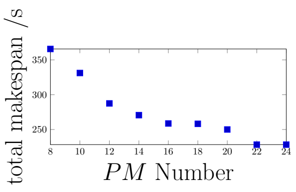
Figure 4.18: Results for the real-world workflows runtime in different node numbers under $M - TS$, $M - S$ and $M - TAA$

Table 4.2: Results of the Real-World DAGs Makespan

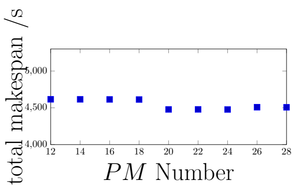
Task	PN	M-TS	M-S	M-TAA	Task	PN	M-TS	M-S	M-TAA
Mon 50	5	138.01	132.23	132.07	Cyb 50	6	585.74	522.03	323.77
Mon 50	8	87.46	87.46	87.28	Cyb 50	10	422.88	416.69	290.62
Mon 50	10	77.12	77.0	76.83	Cyb 50	12	410.19	380.13	262.74
Mon 50	15	66.89	66.43	66.27	Cyb 50	15	313.83	342.87	262.74
Epi 46	4	16584	16585	13393	Si 60	10	7058	6712	4643
Epi 46	5	12455	12455	11469	Si 60	12	4649	4640	4642
Epi 46	7	12226	12234	10672	Si 60	15	7056	6710	4640
Epi 46	10	7744	7744	7728	Si 60	18	4648	4640	4640
Ins 50	5	3354	3319	2905	Ins 50	8	2054	2021	1939
Ins 50	7	2386	2372	2186					
Ins 50	12	1410.8	1410.8	1410.8					



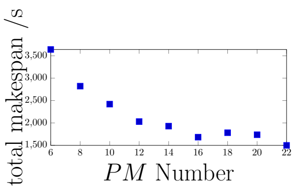
(a) Epigenomic 100



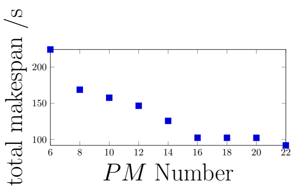
(b) CyberShake 100



(c) Sipht 100

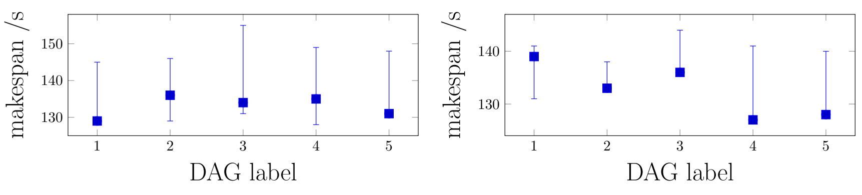


(d) Inspiral 100



(e) Montage 100

Figure 4.19: Makespan-TAA with different number of PMs



(a) Task number = 30, PM = 6 (b) Task number = 30, PM = 8

Figure 4.20: Makespan results for Randomly generated DAGs

the makespan results of the DAGs: The square mark indicates the sequential execution makespan, the dash above the square mark indicates the time-sharing makespan while the dash below the square mark indicates the TAA makespan. The TAA makespan of $DAG1$ and $DAG5$ of 4.20(a), as well as $DAG2$, $DAG3$, $DAG4$ and $DAG5$ of 4.20(b) are the same as their corresponding deadline (i.e. sequential execution makespan). There exits a gap between time-sharing makespan and sequential execution makespan in all cases. In addition, the experiment result indicates that TAA can efficiently reduce the DAGs' time-sharing makespan.

4.8 Summary

In this chapter, we investigated the impact of the time-sharing execution on the DAG makespan. The makespan model in the time-sharing execution was proposed. Based on the makespan model, a Task Migration Algorithm and a Task Allocation algorithm are developed, aiming to reduce the actual makespan of the DAG schedule when the DAG is executed in time-sharing in reality. We conduct the extensive experiments with the real-world workflows. The experimental results show that there exists gap between the makespan in sequential execution, the makespan in time-sharing execution and the makespan obtained by our DAG scheduling algorithm designed for time-sharing systems.

Only considering the time-sharing execution can not result in the accurate estimation of the task makespan. Resource interference is another important factor that influences the performance of co-running tasks in multi-core computers. Our studies show that a task may have different levels of performance degradation when co-running with different tasks. The performance degradation varied from 0% to 80% in our studies. In the next chapter, the scenario that is more practical and complex is assumed. We investigate the impact of these influential factors and predict the performance impact of the co-running tasks on multi-core computers.

CHAPTER 5
Contention-aware Prediction for Performance Impact of Task Co-running in Multi-core Computers

Resource interference is another influential factor to the performance of co-running tasks in multi-core computers. In the task scheduling, it is often assumed that the scheduler knows the execution time of the tasks, based on the assumption that the techniques are present to predict the performance of tasks. However, it is a non-trivial task to product the accurate the performance prediction for tasks, although a number of techniques are indeed developed to predict the task performance [57].

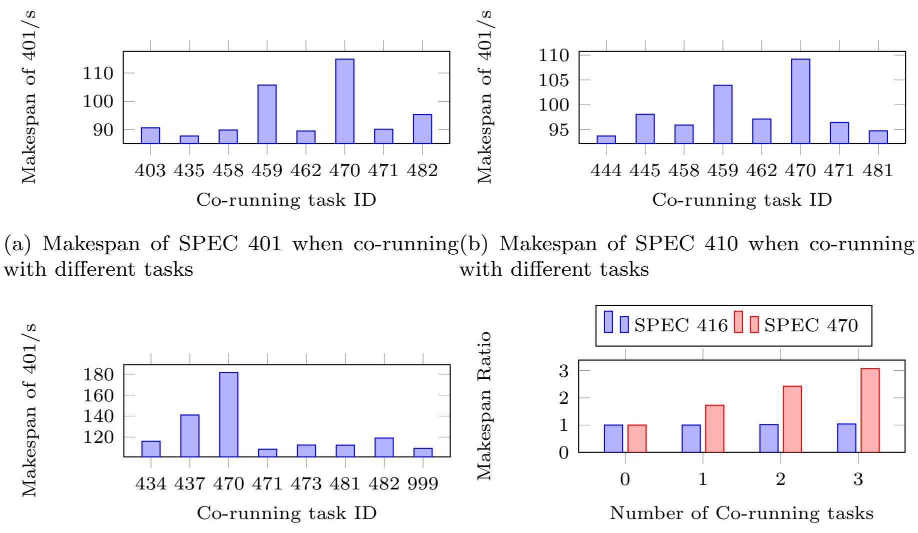
In this chapter, we investigate the influential factors that impact on the performance when the tasks are co-running on multi-core computers. Further, we propose a machine learning-based prediction framework to predict the performance of the co-running tasks. In particular, two prediction frameworks are developed for two types of task in our model: repetitive tasks (i.e., the tasks that arrive at the system repetitively) and new tasks (i.e., the task that are submitted to the system the first time), the difference between which is that we have the historical running information of the repetitive tasks while we do not have the prior knowledge about new tasks. Given the limited information of the new tasks, an online prediction framework is developed to predict the performance of co-running new tasks by sampling the performance events on the fly for a short period and then feeding the sampled results to the prediction framework. We conducted the extensive experiments with the SPEC2006 benchmark suite to compare the effectiveness of different machine learning methods considered in this chapter. The results show that our prediction model can achieve the good enough accuracies for repetitive tasks and new tasks, respectively.

5.1 Motivation and Background

In this section, we conduct experiments to reveal the performance impact problem. The experimental results show that 1) a task may have various performance degradation when co-running with different tasks; 2) The performance degradation of a task that running with different frequencies is non-linear; 2) A task will have varying performance impact when time-sharing executing with different tasks. However, the performance impact is hard to predict by using specific formulation. We use SPEC2006 to conduct the benchmarking experiments to investigate the impact of task co-running on performance.

5.1.1 Performance Impact of Multi-core

Figure 5.1 shows the execution time of three benchmarks in SPEC2006, 401 (401.perlbench is a compression program), 410 (410.bwaves simulates blast waves in three dimensional transonic transient laminar viscous flow) and 470 (470.lbm is a computational fluid dynamics program using the lattice boltzmann method) [36], when they co-run with other SPEC2006 benchmarks on a multi-core processor. In Figure 5.3(a), the execution time of the solo-run of SPEC 401 (i.e., when the benchmark runs on a core without other programs co-running on other cores) is 89.4 seconds. When co-running with other benchmarks, the execution time of 401 vary from 89.50 (co-running with 462) to 114.92 (co-running with 470). Its performance degradation is noticeable: from 0% to 28%. The same phenomenon occurs with other benchmarks in SPEC. Through the experiment, we also observed that some benchmarks, such as SPEC 470 are more contention-sensitive (up to 65%) than others, such as SPEC 444 (up to 5%). In Figure 5.3(d), experiments are conducted on a quad-core processor to show the performance impact of SPEC 416 (416.gamess is a wide range of quantum chemical computations) when it is running with different degrees of contentions. The x-axis represents the number of co-running tasks. For example, x-axis '0' means that SPEC 416(470) is solo-running with no resource contention; x-axis '3' means



(a) Makespan of SPEC 401 when co-running with different tasks

(b) Makespan of SPEC 410 when co-running with different tasks

(c) Makespan of SPEC 470 when co-running with different tasks

(d) Task running with vary contentions

Figure 5.1: Motivation Experiments

that there are four SPEC 416(470) tasks co-running on different cores within the quad-core processor. The y-axis represents the ratio of $Makespan_{co-run}$ to $Makespan_{solo}$. We can see that the resource contention leads to an enormous increasing of completion time of SPEC 470; While there is not any performance impact on SPEC 416 during the co-running.

Almost all latest researches cite cache miss and memory bandwidth as the most important factors that affect the performance of co-running tasks. Based on these two factors, several performance models are constructed to formulate the impact of task co-running on multi-core processors [19][20][112][11]. However, our research show that more factors show noticeable impact on the co-running performance, such as branch-misses, context-switches, and minor faults, etc. We collect 30 performance events provided by the Operating System during the execution of co-running tasks, as shown on the x-axis of Figure 5.2. Figure 5.2 shows the ratio of the values of these performance events gathered when SPEC 459 co-runs with 470 to those when SPEC 459 solo-runs. As can be seen from this figure, the values of the performance events have considerable

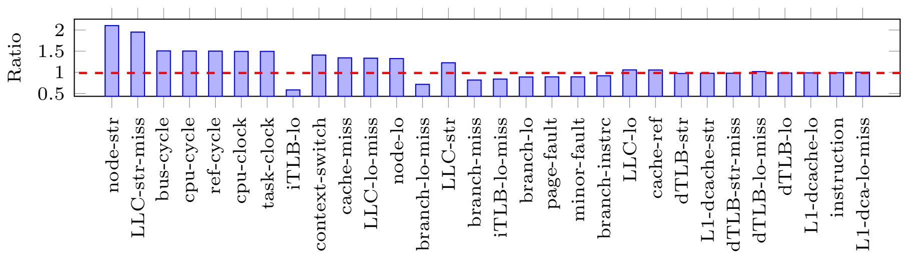


Figure 5.2: Comparison of performance events of SPEC 459 between solo execution and co-running

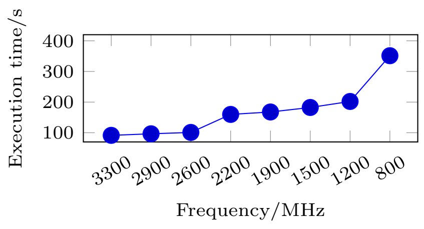
changes when the benchmarks co-run.

5.1.2 Performance Impact of Scaling Frequency

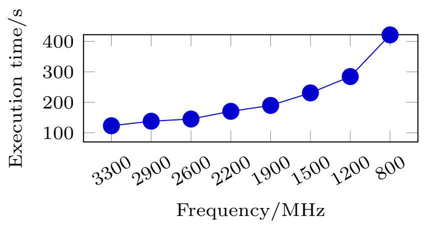
Figure 5.3 shows the execution time of SPEC 403, 437, 450, 462, 470 and 481 when executing under various frequencies. The experimental results show that the relationship between execution frequency and performance degradation is non-linear. From the frequency of 3300 MHz to 800 MHz, the increase in the task execution time vary from 200% to 300%. According to the energy consumption function [1][90], reducing the execution frequency can save overall energy consumption. However, reducing the execution frequency will lead to an increasing of the execution time. It is a popular topic to analyze the trade-off between execution time and energy consumption. Much research make efforts on reducing the energy consumption while satisfying the task's deadline constraint. Predicting the execution time accurately under various frequencies is critical for the scheduler to determine the appropriate execution frequency of a task.

5.1.3 Performance Impact of Time-sharing Execution

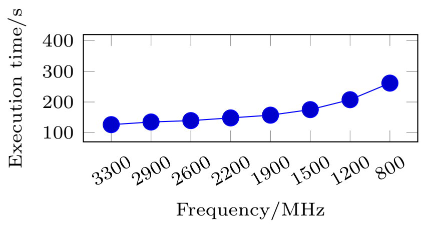
Little research takes into account the time-sharing execution when making scheduling decisions and calculating the completion times of tasks. Figure 5.4(a),5.4(b) show the sequential execution and the time-sharing (concurrent) execution, respectively. Time-sharing techniques can help improve the process-



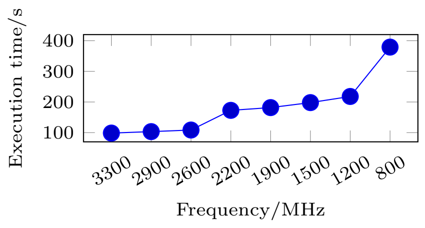
(a) SPEC 403



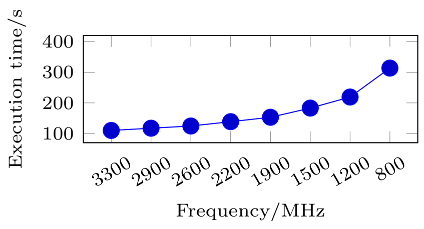
(b) SPEC 437



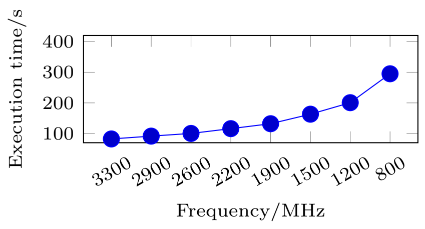
(c) SPEC 450



(d) SPEC 462

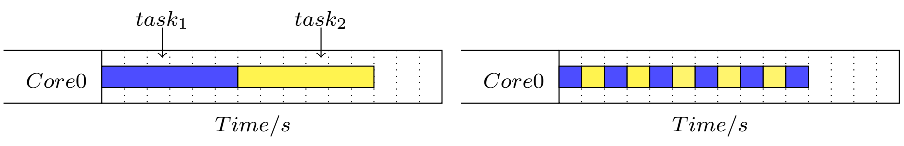


(e) SPEC 470



(f) SPEC 481

Figure 5.3: Execution time for SPEC 403, 462 and 470 when executing under different frequencies



(a) Non-time-sharing executing process (b) Time-sharing executing process

Figure 5.4: Diagrams of two kinds of execution manners

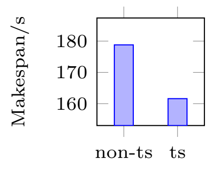
ing efficiency. For example, the solo execution time of SPEC 401 and 429 are 89.409 seconds and 91.469 seconds, respectively. Experimental results show that the makespan of these two tasks under the time-sharing execution is 142.42 seconds, less than the total execution time of 180.6 seconds under the sequential execution. This means that the time-sharing execution reduces the execution time by 21.1%.

We conducted the experiments with 410 combinations of different SPEC benchmark programs running concurrently on the same core. Figure 5.5 presents some results of our experiments. The figures show that the time-sharing execution can reduce the execution time by 4%-55%. Some combinations of concurrently running benchmarks can reduce the makespan significantly while others do not show as much benefit. Thus, it is important to predict the makespan of the time-sharing tasks accurately so that the scheduler can make better decisions as to which tasks should be allocated to the same core.

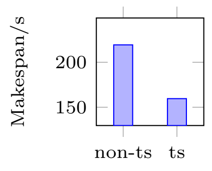
5.2 The Performance Prediction Framework for Co-running Tasks

We develop the performance prediction framework for both repetitive tasks, which has been run in the system before and therefore we have the historical performance event data when the task solo-runs, and new tasks, which are submitted to run on the system the first time.

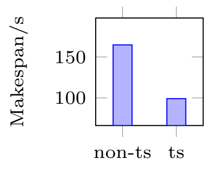
When the co-running tasks are the repetitive tasks, we use the historical performance event data of individual tasks (their solo-run performance data) as



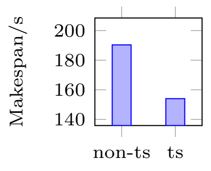
(a) 401&401



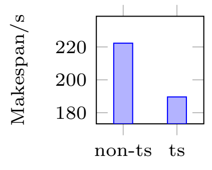
(b) 401&454



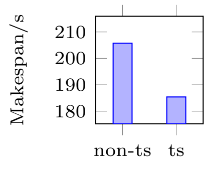
(c) 403&482



(d) 410&433



(e) 410&444



(f) 435&444

Figure 5.5: Non-time-sharing vs Time-sharing makespan (Under the same frequency, 3301 MHz)

the input of the prediction framework. When the co-running tasks contain new tasks, we do not have the prior knowledge about the new tasks and therefore different procedure is developed in this chapter for new tasks.

The prediction framework comprises the following four steps.

1) It runs a series of benchmarks and collect training data on their performance events. The benchmarks are run under all scenarios (i.e. co-running, frequency scaling or time-sharing). The benchmarking details will be presented in next subsection.

2) With the training data, it generates individual prediction model for each scenario using a specific machine learning approach.

3) When a task is to be scheduled, it first recognizes the scenario the task belongs to (i.e. co-running, frequency scaling or time-sharing) and what type of task it is (i.e. repetitive task or new task). Then we use the corresponding prediction model to estimate the execution time of the task.

4) The prediction results of performance impact is fed into the schedulers for making better scheduling decisions.

5.2.1 Feature Selection

The section describes the performance events considered by our prediction approach. The data of all 30 performance events can be collected using Perf (a profiling tool in Linux) during the execution of the tasks. We introduce a notion called the *the rate of performance event*, which equals to the collected value of a performance event divided by the execution time of the task (i.e., the frequency at which the performance event occurs during the execution of the task). We then use the rates of the performance events as the attributes of the training model.

The performance events we collected are as follows [24]:

– Branch-misses: Branch mispredicted/not predicted; Counts the number of executed branches which are mispredicted or not predicted;

– Branch-loads: Branches or other changes in the program flow that could have been predicted by the branch prediction resources of the processor

– Bus-cycles: Bus cycle counter

– Cache-misses: Data read or write operation that causes a refill at (at least) the lowest level of data or unified cache

– Cache-references: Data read or write operation that causes a cache access at (at least) the lowest level of data or unified cache

– Cpu-cycles: Cycle counter

– Instructions: Instruction architecturally executed

– Ref-cycles: Total cycles; not affected by CPU frequency scaling

– Context-switches: Count the number of the process that storing the state of a process (or thread)

– Minor-faults: The code (or data) needed is actually already in memory, but it isn't allocated to that process

– L1-dcache-load-misses: Data read or write operation that causes a refill at (at least) the lowest level of data or unified cache.

– L1-dcache-loads: Data read or write operation that causes a cache access at (at least) the lowest level of data or unified cache.

– LLC-store-misses: Level 2 data cache refill

– LLC-stores: Level 2 data cache access

– dTLB-load-misses: Data read or write operation that causes a TLB refill at (at least) the lowest level of TLB

– iTLB-load-misses: Instruction fetch that causes a TLB refill at (at least) the lowest level of TLB.

– Node-loads: Measure local memory accesses

5.2.2 The Performance Prediction Framework for Repetitive Tasks

The performance impact of a task is defined as the ratio of co-running completion time to its solo completion time. When task t_i and t_j co-run, the predicted performance impact of t_i, denoted by PI_T_i, is represented as in 5.1, where PE_i and PE_j are the set of solo-run performance events of task t_i and t_j, respectively (the value of a performance event is the rate of performance event, namely the counter of the performance event divided by the length of the period in which the event is collected); $\Gamma_{co-running}$ is a trained model based on a set of parameters including normalization (if the input data is normalised), discretization (whether discretizing the data and with specific number of bins), shuffle (if the data is going to be sampled), predictor (a selection from four supported machine learning algorithms such as regression, Naive Bayes, SVM and Random Forest), predType (two types of prediction that are regression and classification).

$$PI\_T_i = \Gamma_{co-running}(PE_i, PE_j) \tag{5.1}$$

Similarly, the execution time with various CPU frequency can be derived by

$$PI\_T_i = \Gamma_{freq}(exec\_freq, PE_{refer}) \tag{5.2}$$

Where function Γ_{freq} is the trained model (still based on the parameters discussed above) concerning the performance impact with various execution frequencies. The execution frequency $exec\_freq$ acts as an important input of the model; PE_{refer} is the reference performance event, i.e., the performance event rate when the task is run with this referenced CPU frequency.

Finally, supposing task t_i and t_j are running on the same core in a time-sharing manner (concurrently), the impact on the performance of task t_i can be derived by

$$PI\_T_i = \Gamma_{ts}(PE_i, PE_j) \tag{5.3}$$

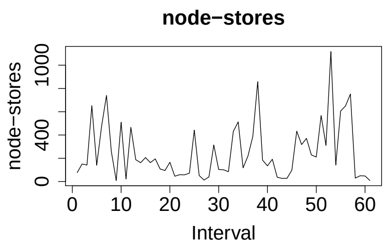
Where Γ_{ts} analyzes the performance impact of the time-sharing execution and produces the makespan of task t_i and t_j.

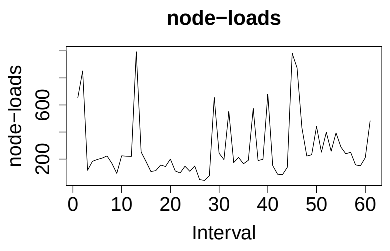
With the history information of the reference execution time of a task, we can derive the interference-aware execution time of the task straightforward.

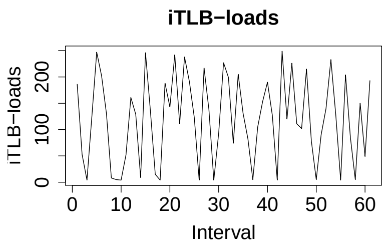
5.2.3 The Performance Prediction Framework for New Tasks

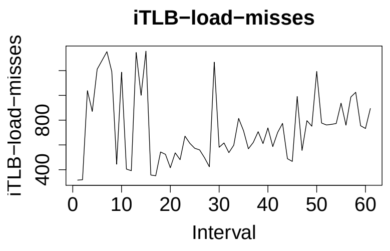
In order to understand why our prediction model for new tasks works, see a benchmarking experiment we conducted. Figure 5.6 shows the trend of the selected performance events of SPEC 401 during its co-running with SPEC 403. In the experiments, we collect the performance event data once every 500ms. The execution time of SEPC 401 (1 iteration) is 30 seconds. Thus we obtained 60 sampled data (time intervals). As can be seen from this figure, many performance events show repeated or similar trend as the co-running tasks progress, which provides a ground for our prediction model for new tasks.

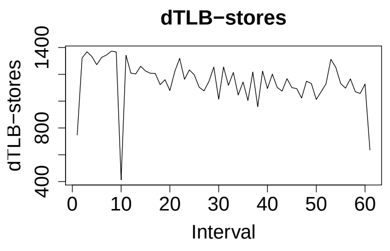
Since we do not have the prior knowledge about the new tasks, we develop a two-stage prediction framework. In the first stage, we construct a *prediction model for each performance event.* Thus we have 30 models in total, corresponding to the 30 performance events. We sample the performance events at a preset sampling rate for a preset period when the task is solo-executing; Then

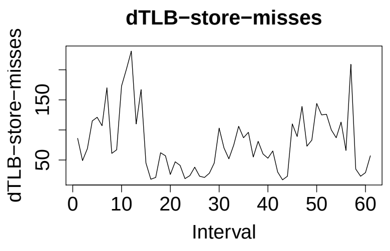

node−stores
node−stores
0
400
1000
0 10 20 30 40 50 60
Interval

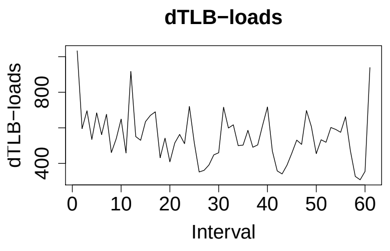

node−loads
node−loads
200
600
0 10 20 30 40 50 60
Interval

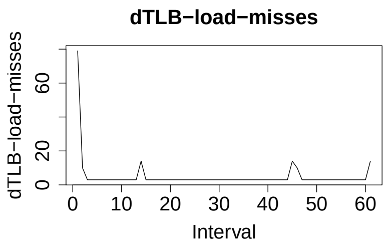

iTLB−loads
iTLB−loads
0
100
200
0 10 20 30 40 50 60
Interval

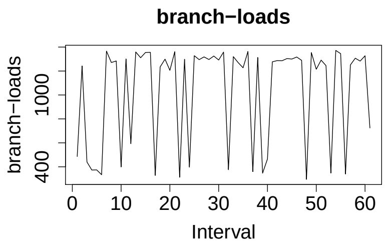

iTLB−load−misses
iTLB−load−misses
400
800
0 10 20 30 40 50 60
Interval

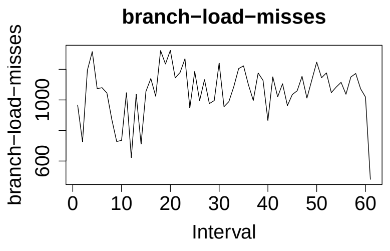

dTLB−stores
dTLB−stores
400
800
1400
0 10 20 30 40 50 60
Interval

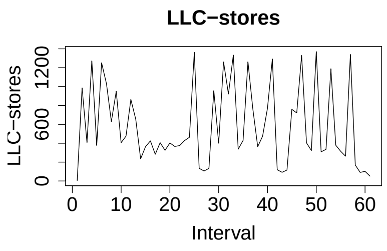

dTLB−store−misses
dTLB−store−misses
50
150
0 10 20 30 40 50 60
Interval

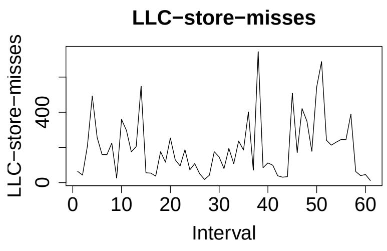

dTLB−loads
dTLB−loads
400
800
0 10 20 30 40 50 60
Interval

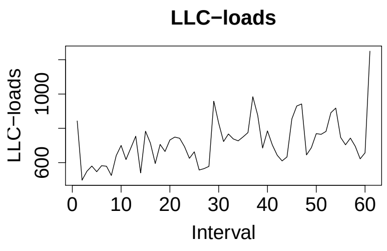

dTLB−load−misses
dTLB−load−misses
0
20
60
0 10 20 30 40 50 60
Interval

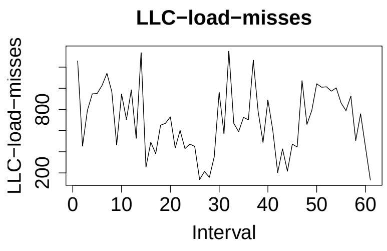

branch−loads
branch−loads
400
1000
0 10 20 30 40 50 60
Interval

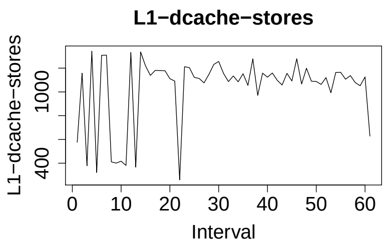

branch−load−misses
branch−load−misses
600
1000
0 10 20 30 40 50 60
Interval

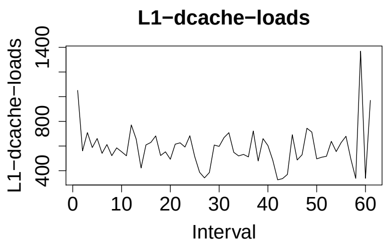

LLC−stores
LLC−stores
0
600
1200
0 10 20 30 40 50 60
Interval

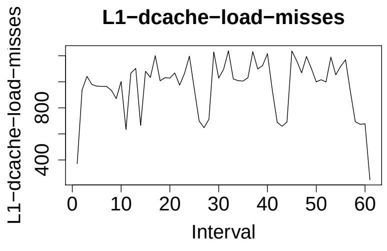

LLC−store−misses
LLC−store−misses
0
400
0 10 20 30 40 50 60
Interval

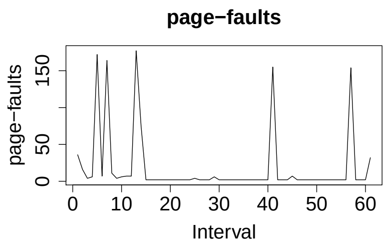

LLC−loads
LLC−loads
600
1000
0 10 20 30 40 50 60
Interval

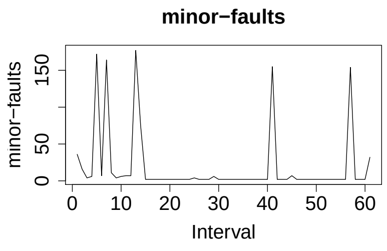

LLC−load−misses
LLC−load−misses
200
800
0 10 20 30 40 50 60
Interval

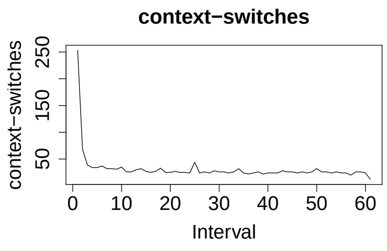

L1−dcache−stores
L1−dcache−stores
400
1000
0 10 20 30 40 50 60
Interval


L1−dcache−loads
L1−dcache−loads
400
800
1400
0 10 20 30 40 50 60
Interval


L1−dcache−load−misses
L1−dcache−load−misses
400
800
0 10 20 30 40 50 60
Interval


page−faults
page−faults
0
50
150
0 10 20 30 40 50 60
Interval


minor−faults
minor−faults
0
50
150
0 10 20 30 40 50 60
Interval


context−switches
context−switches
50
150
250
0 10 20 30 40 50 60
Interval

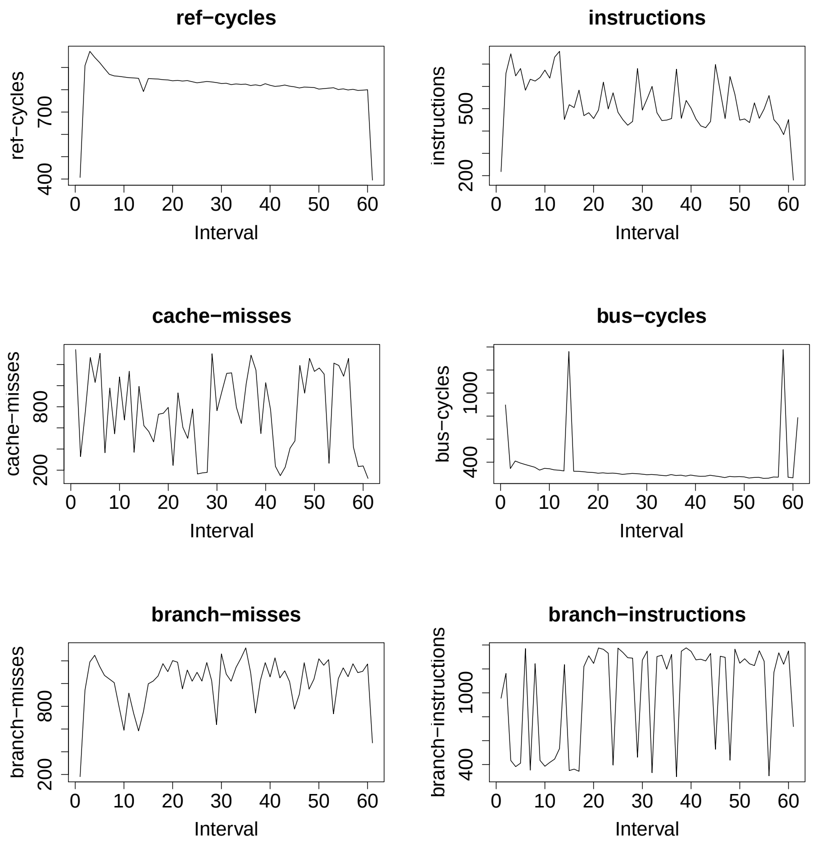


Figure 5.6: The trend of performance events as the co-running tasks progress

we sample the performance events for the same length of period when the task is co-running with other tasks. For example, the performance events are sampled once every 200 ms for 2 seconds for a task of 100 seconds. The prediction model for performance event takes the sampled data of the performance event as input and predicts the value of the performance event when the solo-running and the co-running task complete, respectively. In the second stage, we use the impact ratio of the performance events to predict the performance impact of the co-running tasks. The impact ratio is defined as the ratio of the value of the co-running performance event to the value of the solo-running performance event that is predicted in the first stage.

The two-stage prediction model is formulated as follows. t_{sp} denotes the time period of sampling and int represents the sampling interval (the inverse of the sampling rate). Then t_{sp}/int is the number of sampled data we obtain for a performance event.

The predicted data of the performance event when the solo-running task (or co-running task) t_i completes, denoted by PPE_i^{solo} (or PPE_i^{co}), can be represented by the vector that is derived from Equation 5.4 (or 5.5). F_1^n represents the prediction model for performance event n. $s\_PE_j^n$ and $c\_PE_j^n$ denote the performance event data of the j-th sampling interval for performance event n under solo-run and co-run, respectively. In our work, there are in total 30 performance events (i.e., $m = 30$). Thus we train 30 models in the first stage.

$$PPE_i^{solo} = [F_1^1(s\_PE_1^1, ..., s\_PE_{t_{sp}/int}^1), ..., F_1^m(s\_PE_1^m, ..., s\_PE_{t_{sp}/int}^m] \quad (5.4)$$

$$PPE_i^{co} = [F_1^1(c\_PE_1^1, ..., c\_PE_{t_{sp}/int}^1), ..., F_1^m(c\_PE_1^m, ..., c\_PE_{t_{sp}/int}^m] \quad (5.5)$$

In the second stage, the impact ratio vector of task t_i, denoted by IR_i, can be derived from:

$$IR_i = \frac{PPE_i^{co} - PPE_i^{solo}}{PPE_i^{solo}} \tag{5.6}$$

Then the prediction model for the performance impact of task t_i can be represented by

$$PI\_T_i = \Gamma'_{co-running}(IR_i) \tag{5.7}$$

where $\Gamma'_{co-running}$ represents the trained model for predicting the performance impact of new tasks. Note that we do not need the performance event data for the co-running task t_j as the input of this formula because the execution information of t_j has been reflected in the sampled performance event data since tasks t_i and t_j are co-running. Furthermore, we can predict the performance impact of a specific task in the same way, no matter how many tasks it is co-running with in a multi-core processor.

In the above model representations, PPE_i^{solo} and PPE_i^{co} represents the first stage work while PI_T_i represents the second stage work in the two-stage prediction model. In the first stage, we construct a *prediction model* for each $performance$ *event*. We sample the performance events at a present sampling rate under the specific frequency. The prediction model for performance event tasks the sampled data of the performance events as input and predicts the value of the performance event when the task complete. In the second stage, we use the outputs of the first stage models to predict the performance impact of the task.

For the scenario of the tasks running under various CPU frequencies, the "performance impact" of a task refers to the ratio of the execution time of the task under a specific CPU frequency to the execution time under the reference CPU frequency (e.g. the highest CPU frequency). For example, suppose that we use the highest CPU frequency (3300 MHz) as the reference CPU frequency, and that task T_i takes 100 seconds to complete under the frequency of 3300

MHz, but spends 300 seconds to complete under 800 MHz. Then the impact of running the task under 800 MHz on its performance is 3.

Similar to the scenario of co-running tasks, the two-stage prediction model is formulated as follows. The predicted data of the performance event when the tasks running under a specific frequency (or under a reference frequency), say f_{spe} (or f_{ref}), can be represented by the vector that is derived from Equation 5.8 (or 5.9). F_1^n represents the prediction model for performance event n. $s\_PE_j^n$ and $r\_PE_j^n$ denote the performance event data of the j-th sampling interval for performance event n under specific frequency f_{spe} and reference frequency f_{ref}, respectively.

$$PPE_i^{f_{spe}} = [F_1^1(s\_PE_1^1, ..., s\_PE_{t_{sp}/int}^1), ..., F_1^m(s\_PE_1^m, ..., s\_PE_{t_{sp}/int}^m)] \quad (5.8)$$

$$PPE_i^{f_{ref}} = [F_1^1(r\_PE_1^1, ..., r\_PE_{t_{sp}/int}^1), ..., F_1^m(r\_PE_1^m, ..., r\_PE_{t_{sp}/int}^m)] \quad (5.9)$$

In the second stage, the impact ratio vector of task t_i, denoted by IR_i, can be derived from:

$$IR_i = \frac{PPE_i^{spe} - PPE_i^{ref}}{PPE_i^{ref}} \quad (5.10)$$

Then the prediction model for the performance impact of task t_i can be represented by

$$PI\_T_i = \Gamma'_{freq}(IR_i) \quad (5.11)$$

where Γ'_{freq} represents the trained model for predicting the performance impact of new tasks. Note that when predicting the performance impact of CPU frequencies, we do not need to know the exact execution time of a task under the reference CPU frequency. Performance impact here is an indicator that helps us to make a trade-off between execution time and energy consumption and to determine the appropriate execution frequency for a task.

5.3 Other Machine Learning Approaches

The machine learning approaches used in our prediction frameworks are: linear regression, naive Bayes, support-vector machine (SVM) and random forest. These four algorithms are the most basic machine learning supervised algorithms and widely used in industry. We examined these four popular machine learning approaches, aiming to identify most effective approach for particular scenarios. These machine learning approaches are explained in this section.

5.3.1 Linear Regression

Linear regression is a linear approach to modelling the relationship between one or more variables. In our work, given a data set $\{PI\_T_i, x_{i1}, x_{i2}, ..., x_{ip}\}_{i=1}^{n}$ of n statistical units, a linear regression model assumes that the relationship between the dependent variable PI_T_i and the p-vector of regressors x is linear. Here, PI_T_i represents the performance impact (to be predicted) and $\{x_1, x_2, ..., x_p\}$ indicates the performance events of the task. This relationship is modelled through a disturbance term or error variable ε – an unobserved random variable that adds "noise" to the linear relationship between the dependent variable and regressors [94]. Thus the model tasks the form:

$$PI\_T_i = \beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip} + \varepsilon_i = x_i^T + \varepsilon_i, i = 1, ..., n, \tag{5.12}$$

where ^{T} denotes the transpose, so that $x_i^T\beta$ is the inner product between vectors x_i and β. β is a $(p+1)$-dimensional parameter vector and β_0 is the intercept term. The estimation in linear regression focuses on β.

5.3.2 Naive Bayes

In the abstract term, Naive Bayes is a conditional probability model: given a problem instance to be classified, represented by a vector $x = (x_1, ..., x_n)$ of n features, $p(C_k|x_1, ..., x_n)$ is assigned to this instance probabilities of k possible outcomes (or called class C_k). In our work, the n features indicates the performance events and the class C_k is a interval of execution time. Using the Bayes' theorem, the conditional probability can be decomposed as:

$$p(C_k|x) = \frac{p(C_k)p(x|C_k)}{p(x)} \tag{5.13}$$

Using the conditional independence assumptions of Naive Bayes [96], the joint model of the n features can be expressed as:

$$p(C_k|x_1, ..., x_n) \propto p(C_k, x_1, ..., x_n) = p(C_k)p(x_1|C_k)p(x_2|C_k)... = p(C_k)\prod_{i=1}^{n} p(x_i|C_k) \tag{5.14}$$

Where $\propto$ denotes the proportionality.

When we predict the execution time of a task, the Naive Bayes classifier calculates all k probabilities and assign the execution time to the most probable class $\hat{y} = C_k$:

$$\hat{y} = \underset{k \in 1,2,...,K}{arg\max}\ p(C_k)\prod_{i=1}^{n} p(x_i|C_k) \tag{5.15}$$

Note that when dealing with the continuous data, we assume that the continuous variables are distributed to a normal distribution (although some data does not satisfy this condition). The probability density of the normal distribution is:

$$p(x = v|C_k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} e^{-\frac{(v-\mu_k)^2}{2\sigma_k^2}} \tag{5.16}$$

Where μ is the expectation of the distribution. σ is the standard deviation and σ^2 is the variance.

5.3.3 Support Vector Machine

The support-vector machine maps the non-linearly input vectors to a high-dimensional feature space. In this feature space, a linear decision surface is constructed, which ensures high generalization ability [17].

5.3.4 Random Forest

Based on the decision tree, the random forest uses the random choice methods to deal with the training data and features. A Random Forest grows many classification trees. Within the forest, each tree is an independent classifier. During the classification process, each tree gives a suggest result, and the forest chooses the result with the most votes among all the trees in the forest.

Each tree is grown in the following way [8]:

1) Suppose there are N pieces of data in the training set. For each tree, N pieces of data are randomly picked using the bootstrap method, from the original data. This sample will be the training set for growing the tree. 2) Suppose there are M features, a constant number $m << M$ is specified such that at each node, m variables are selected randomly from M and the best split of these m variables is used to split the node. We recommend $m = \sqrt{M}$ for classification and $m = M/3$ for regression. 3) Each tree is grown as much as possible. There is no pruning.

Random forest overcomes the weakness that the single decision tree is easily influenced by the noise data. Bootstrap aggregating helps reduce over-fitting.

5.4 Evaluation

The accuracy of our prediction frameworks plugged with the above four machine learning approaches is evaluated in this section. The testing environment is a Personal Computer with a 3.30 GHz dual-core Intel i5 CPU and 8 GB memory.

It has 32k L1d cache, 32k L1i cache, 256k L2 cache and 6144k L3 cache. 28 benchmarks used in the evaluation come from the Standard Performance Evaluation Corporation (SPEC) benchmark suite. There are For each application, the features are collected only once statically. These data are repeatedly used for training and predicting. Note that the training time of our models vary from 2.1 to 4.6 seconds and the prediction time of one instance is very short and can be neglected. Note that all the results we present in this section is based on an average value of several runs.

5.4.1 Experiment Results for Co-running Tasks

We co-run the benchmarks in SPEC on two cores. There are in total $\binom{28}{2} = 406$ combinations. Each combination generates two instances. Therefore, we obtained 812 pieces of training data. We choose a 4:1 split for training and testing. Table 5.1 shows the experiment results for predicting the performance impact of co-running repetitive tasks and new tasks. We divide our experiments into four categories: Static regression and static classification are for repetitive tasks while online regression is for new tasks.

Figure 5.7 and 5.8 are the residual analysis of the trained model. 1) In the "Residual vs. Fitted" figure, the y axis is the residual while the x axis is the fitted values. The residuals "bounce randomly" around the 0 line. This suggests that the assumption that the relationship is linear is reasonable. There are several outliers that need to be removed from the training set, such as 647, 619 and 707, etc. 2) In the "Normal Q-Q" figure, the results suggest that the residuals (and hence the error terms) are normally distributed but with the several outliers. 3) The "scale-location" plot shows whether the residuals are spread equally along the ranges of predictors. Here, we can see a horizontal line with equally (randomly) spread points, which suggests that the two trained models satisfy homoscedastic. 4) the "Residuals vs Leverage" plot helps to find the influential cases if any. The results for the static model are to be expected except there exit several potential problematic cases in the training set of the

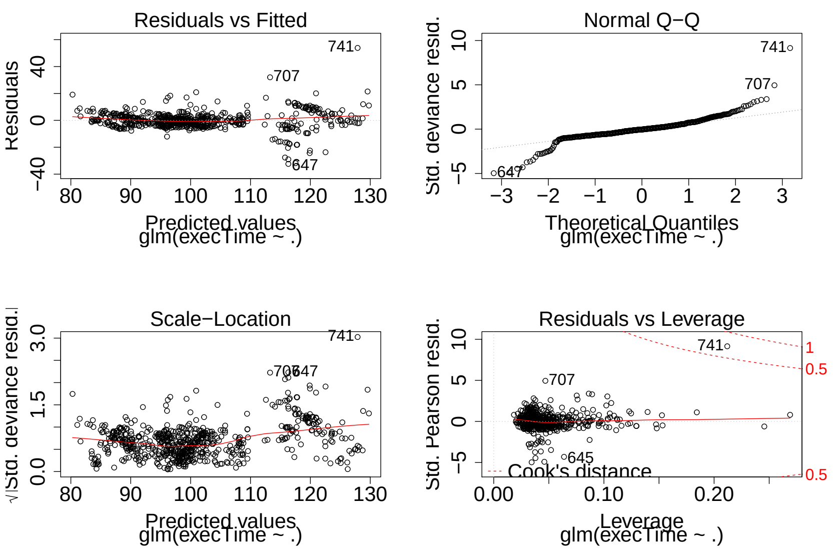


Figure 5.7: Residual analysis for co-running task prediction (online)

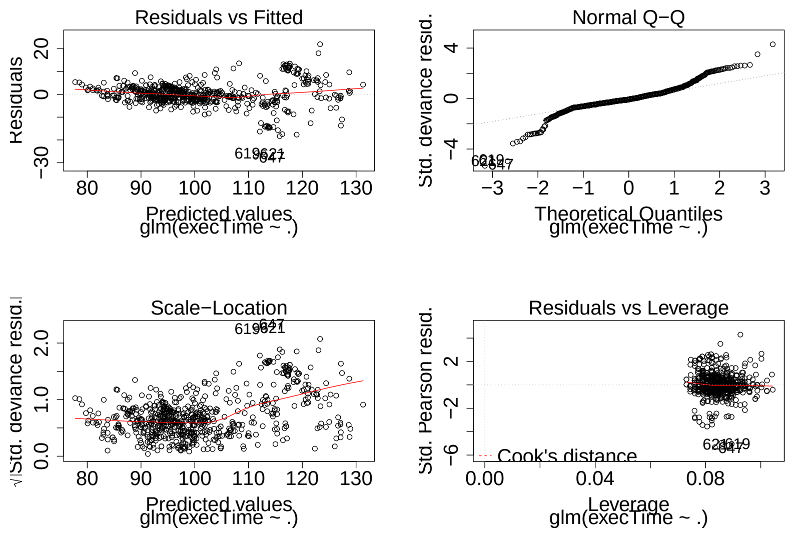


Figure 5.8: Residual analysis for co-running task prediction (static)

Predictors	**Static Reg**			**Static Classifi**		
	d&n	**n-d**	**n-n**	**d&n**	**n-d**	**n-n**
Regression	87.65%	70.99%	86.42%	87.04%	81.48%	83.59%
Naive Bayes	–	–	–	70.20%	66.89%	68.21%
SVM	98.15%	90.12%	96.91%	86.06%	78.15%	76.16%
Random Forest	99.38%	96.68%	98.77%	92.05%	–	94.04%
Predictors	**Online Reg**			**Online Classifi**		
	d&n	**n-d**	**n-n**	**d&n**	**n-d**	**n-n**
Regression	63.58%	60.49%	62.96%	85.19%	81.90%	86.42%
Naive Bayes	–	–	–	58.06%	67.74%	58.06%
SVM	44.87%	43.59%	44.87%	82.80%	74.19%	82.80%
Random Forest	93.59%	93.59%	94.23%	87.10%	–	83.87%

Table 5.1: Experiment results for predicting the co-running time for repetitive tasks and new tasks, respectively

online model (with the row numbers of the data in the dataset).

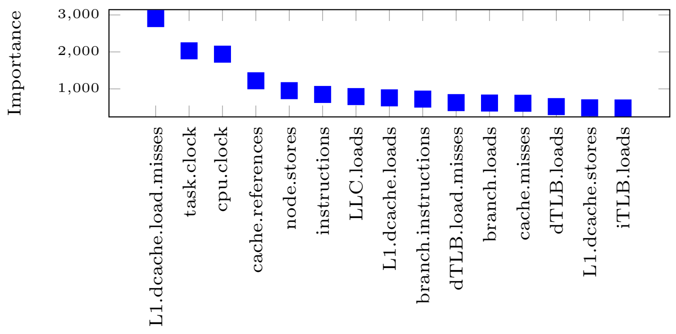
The predicting result for the regression is a number while the predicting result for classification is a range. For the regression, we set a tolerance of 3%. Namely, if the difference between the predicting result and the actual measurement is less than 3%, we regard the predicting result as being correct. We set this tolerance because the measured co-running time is not constant. The execution time of a specific task fluctuates even when it co-runs with the same task. For the classification, we set the reasonable ranges for the data. The difference between the upper bound and lower bound is around 3% of the average performance impact of the application. If the actual performance impact resides within the range that we predict, we regard the predicting result as being correct.

In Table 5.1, we observe that our model can predict the co-running time accurately for most applications. Furthermore, among the predictors, SVM and random forest have the best accuracy for both static and online tests. Random forest achieves over 90% accuracy in both regression and classification for repetitive tasks. We also preprocess the data by discretizing and normalizing the data (denoted by d&n), comparing with the prediction accuracies in the cases of non-discretizing (denoted by n-d) and non-normalizing (denoted by n-n). We discretizes the attributes (i.e. performance event) into specific number of *bins*. This operator discretizes the selected numerical attributes to nominal attributes. To achieve this, a *width* is calculated from $(max\text{-}min)/bins$. *max* and *mix* represent the maximum and the minimum of each attribute, respectively. The range of numerical values is partitioned into segments of equal size (i.e. *width*). Then we assigned the attributes to the corresponding segments. Discretization can help to reduce the categories to a reasonable number that the classifier is able to handle.

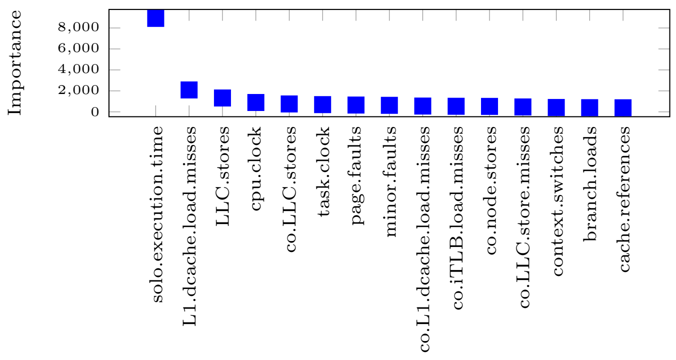
Comparing to the predicting result with non-discretizing and non-normalizing, the discretization and normalization performs a better accuracy in most cases. Discretized feature has strong robustness to the abnormal data. It makes the prediction model more stable and reduces over-fitting. Normalization accelerates the gradient descent and also achieves a higher accuracy, especially for SVM and linear regression. In the online prediction model (for new tasks), we did not normalize the data because the input data are ratios.

Furthermore, we sort the features by their importance scores under the random forest predictor. The IncNodePurity reflects the total decrease in node impurities from splitting on the variable, averaged over all trees. We find the features which are important for the prediction. In the order of their decreasing importance scores, these features are instructions, bus cycles, LLC loads, L1 dcache loads, dTLB loads, branch instructions, branch.loads and context switches. These features are essential for the prediction of performance impact because the prediction accuracy drops significantly (79.3% and 71.79% for static prediction and online prediction, respectively) when we remove any of these features from the feature set of the input data.

Figure 5.9 shows the 15 most important features when predicting the execution time of co-running tasks. Figure 5.9(a) shows the feature importance for online predicting. In this scenario, we do not have any direct information of the co-running tasks and all resource interferences are reflected by the



(a) online predicting



(b) static predicting

Figure 5.9: Top 15 important features for predicting the execution time of co-running tasks

sampling data of the profiled task itself. In this case, the most important feature is "L1.decache.load.misses", followed by "task.clock", "cpu.clock" and "cache.references", etc. The feature CPU clock represents the total time spent on the CPU and task clock indicates the time particular spent on the profiled task. Figure 5.9(b) shows the feature importance for static predicting. In this scenario, we have the history information of the tasks and do not need to do online sampling. In this case, the most important feature for predicting is "solo.execution.time", followed by "L1.decache.load.misses", "LLC.stores", "CPU.clock" and "co.LLC.stores". The "LLC.stores" represents the last level cache access of the profiled task while the "co.LLC.stores" represents the last level cache access of the co-running task. These features are important because they can reflect the resource contention of the co-running tasks.

5.4.2 Experiments with Various CPU Frequencies

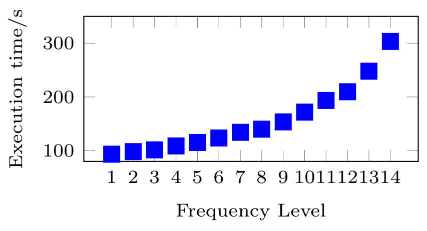
Using the similar way, we train our model to predict the execution time under various CPU frequencies. The input of our model is the execution frequency of a task and the performance event data of the task. In our experimental platform, the CPU frequency ranges from level 0 (highest frequency) to level 14 (lowest frequency), which are 3301 MHz, 3300 MHz, 2900 MHz, 2800 MHz, 2600 MHz, 2400 MHz, 2200 MHz, 2000 MHz, 1900 MHz, 1700 MHz, 1500 MHz, 1300 MHz, 1200 MHz, 1000 MHz and 800 MHz.

We collected the execution times of SPEC and NPB under all frequencies, which generates around 500 pieces of data in our training set. There is a big gap between the task execution time under the highest and the lowest frequency. Different tasks show different degrees of sensitivity to the scaled CPU frequencies. For example, SPEC 401 takes 89.85 seconds to complete under 3300 HMz and 303.34 seconds to complete under 800 MHz; The completion time of SPEC 998 is 97.91 seconds under 3300 HMz while it is 268.41 seconds under 800 MHz. Further, the relationship between frequency and execution time is non-linear. Figure 5.10 shows some examples of the execution time of SEPC benchmarks executing under various frequencies. Benchmarks such as SEPC 401 and 410 are CPU frequency sensitive. When the CPU frequency is scaled down, the execution time increases dramatically by 200%. While benchmarks such as SPEC 429 and 450 are less sensitive to the variation of CPU frequencies. The execution times of these two benchmarks increase by around 92% and 100%, respectively, when the frequency changes from the highest to the lowest level. SPEC 998 and 999 have a similar relationship between CPU frequency and execution time (as shown in figure 5.10(g) and 5.10(h)).

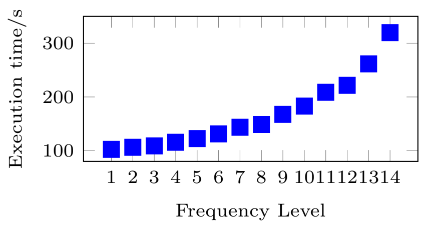
Figure 5.11 and 5.12 show the residual results of the trained model. 1) In the "Residual vs. Fitted" plot, as defined, the residuals appear on the y axis and the fitted values appear on the x axis. The residuals "bounce randomly" around the 0 line. This suggests that the assumption that the relationship is linear is reasonable. There are several outliers that need to be removed from training set, such as 394, 408 and 410, etc. 2) The "Normal Q-Q" plots suggest that the residuals (and hence the error terms) are normally distributed but with the several outliers. 3) The "scale-location" is good because we see a horizontal line with equally (randomly) spread points. 4) The "Residuals vs Leverage" plot of static model looks fine but there exits several potential problematic cases in the training set of the online model (with the row numbers of the data in the dataset).

We can observe from table 5.2 that the execution time under various frequencies can be predicted accurately by machine learning approaches. SVM and random forest produce very high accuracy of 98.81% and 97.62%, respectively. Similar as in the former subsection, discretization and normalization generate better accuracy in most cases, comparing to non-discretizing and non-normalizing predicting result.

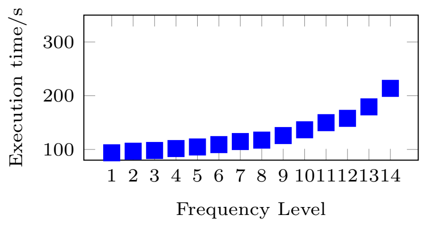
Figure 5.13 shows the 15 most important features for predicting execution time under various frequencies. For static prediction, the most important feature that can achieve the best available split is frequency. In this case, the input performance events for a specific task are the same (due to the property of



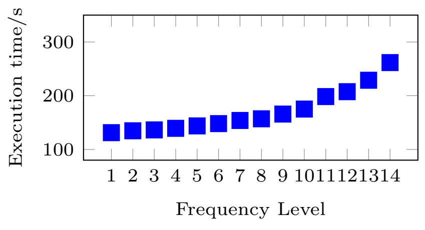
(a) SPEC 401



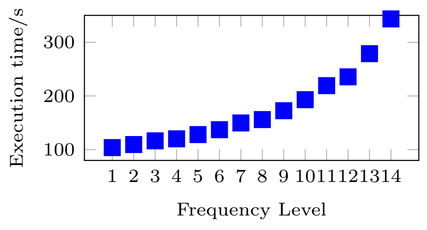
(b) SPEC 410



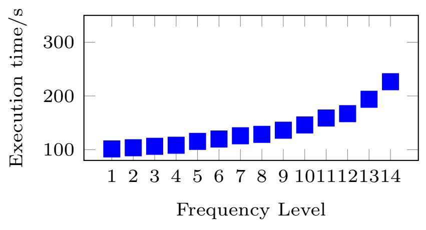
(c) SPEC 429



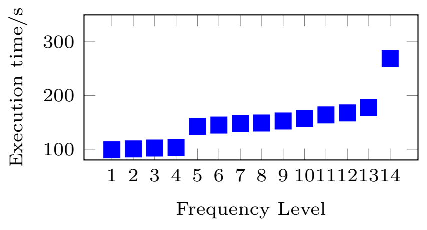
(d) SPEC 450



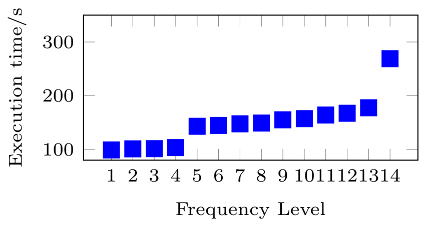
(e) SPEC 471



(f) SPEC 482



(g) SPEC 998



(h) SPEC 999

Figure 5.10: Execution time for SPEC benchmarks when executing under different frequencies

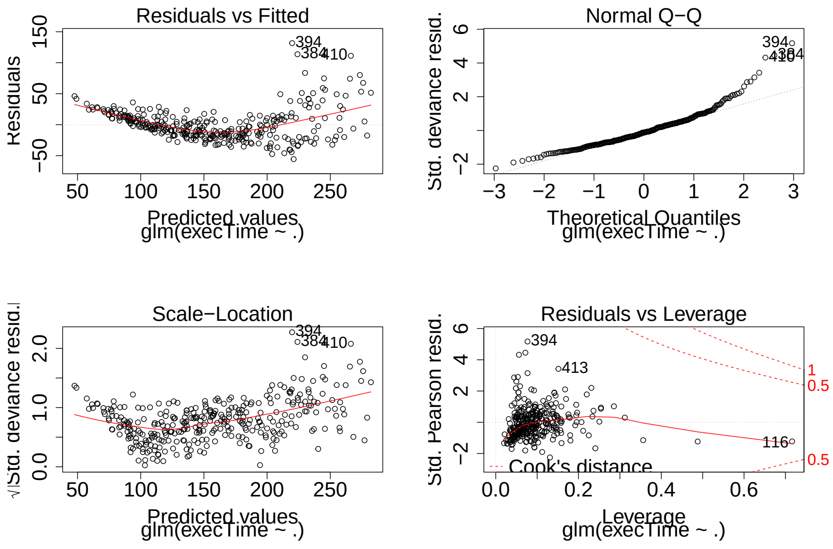


Figure 5.11: Residual analysis for execution time prediction with various frequencies (online)

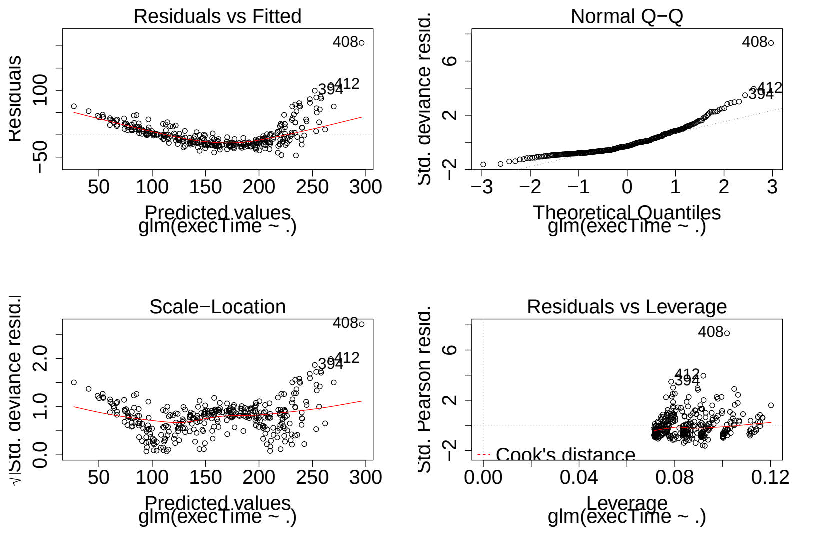
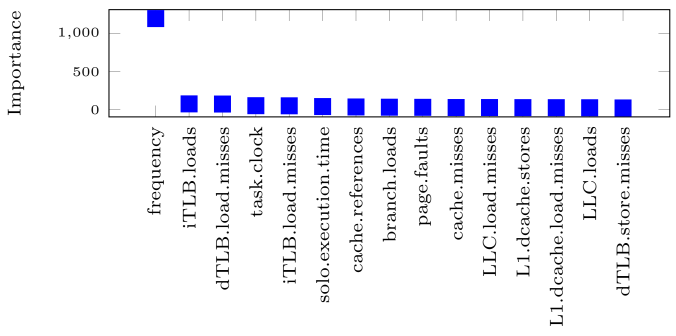


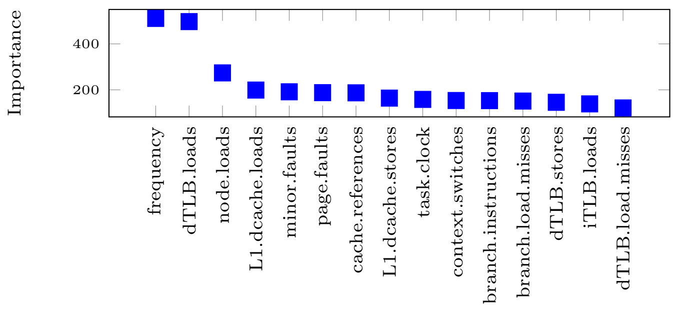
Figure 5.12: Residual analysis for execution time prediction with various frequencies (static)

Predictors	**Static Reg**			**Static Classifi**		
	d&n	**n-d**	**n-n**	**d&n**	**n-d**	**n-n**
Regression	91.67%	76.19%	72.62%	86.90%	73.81%	77.38%
Naive Bayes	–	–	–	48.19%	45.78%	56.63%
SVM	96.43%	95.24%	91.67%	61.45%	50.60%	53.01%
Random Forest	97.62%	96.43%	95.24%	77.11%	–	67.47%
Predictors	**Online Reg**			**Online Classifi**		
	d&n	**n-d**	**n-n**	**d&n**	**n-d**	**n-n**
Regression	72.62%	73.81%	65.48%	90.48%	85.71%	73.81%
Naive Bayes	–	–	–	69.88%	68.67%	65.06%
SVM	98.81%	97.62%	88.10%	77.11%	61.45%	73.49%
Random Forest	96.43%	97.62%	94.05%	81.93%	–	80.72%

Table 5.2: Predicting the execution time under various frequencies for repetitive tasks and new tasks, respectively



(a) static predicting



(b) online predicting

Figure 5.13: Top 15 important features for predicting the execution time under various frequencies

Predictors	**Static Reg**			**Static Classifi**		
	d&n	**n-d**	**n-n**	**d&n**	**n-d**	**n-n**
Regression	62.96%	57.41%	62.35%	60.94%	58.02%	56.79%
Naive Bayes	–	–	–	35.19%	40.12%	36.42%
SVM	82.72%	77.93%	76.54%	51.85%	47.53%	% 48.77
Random Forest	87.04%	85.78%	86.42%	59.61%	–	56.17%

Table 5.3: Predicting the time-sharing makespan for repetitive tasks

static prediction). The only feature that can distinguish different data pieces of the same task is 'frequency'. For online prediction, several features such as frequency, dTLB.loads, node.loads and L1.dcache.loads (etc.) play important roles in the prediction model.

5.4.3 Experiment Results for Time-sharing Tasks

Table 5.3 shows that our model can predict the execution times under the time-sharing execution accurately for most applications. Furthermore, among the predictors, SVM and random forest have the better accuracy. Random forest achieves nearly 90% accuracy in both regression and classification for repetitive tasks. We also preprocess the data by discretizing and normalizing the data (denoted by d&n), comparing with the prediction accuracies in the cases of non-discretizing (denoted by n-d) and non-normalizing (denoted by n-n). Discretized feature has strong robustness to the abnormal data. It makes the prediction model more stable and reduces over-fitting.

Figure 5.14 is the residual analysis of the trained model. 1) In the "Residual vs. Fitted" figure, the y axis is the residual while the x axis is the fitted values. The residuals "bounce randomly" around zero. This suggests that the assumption that the relationship is linear is reasonable. There are several outliers that need to be removed from the training set, such as 654 and 715, etc. 2) In the "Normal Q-Q" figure, the results suggest that the residuals (and hence the error terms) are normally distributed but with the several outliers. 3) The "scale-location" plot shows whether the residuals are spread equally along the

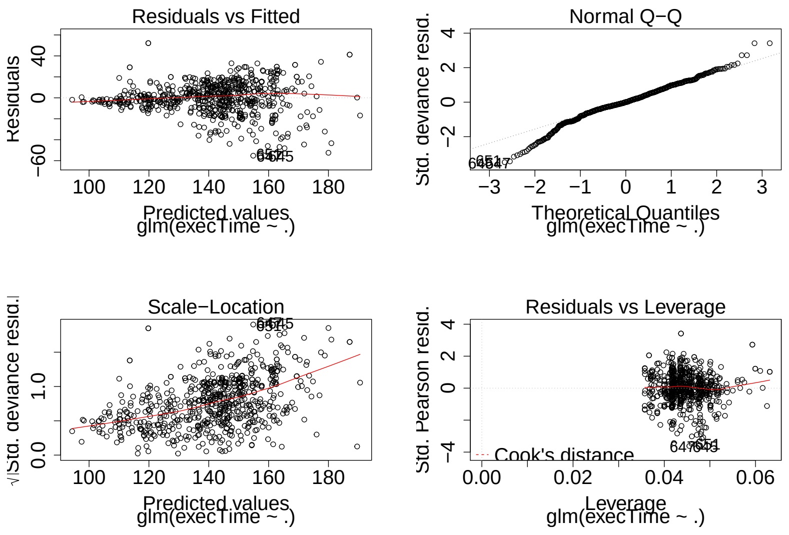


Figure 5.14: Residual analysis for execution time prediction time-sharing executing tasks (static)

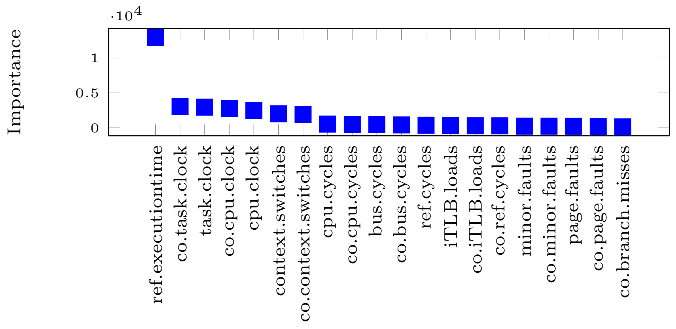


Figure 5.15: Top 20 important features for predicting the execution time of time-sharing tasks

ranges of predictors. Here, we can see a horizontal line with equally (randomly) spread points, which suggests that the two trained models satisfy homoscedastic. 4) the "Residuals vs Leverage" plot helps find the influential cases if any. The results for the static model are to be expected except that there exit several potential problematic cases (with the row numbers of the data in the dataset).

Figure 5.15 shows the 20 most important features for predicting execution. The most important feature is the reference execution time of the task. Followed by the feature "task clock" and "CPU clock". The feature "CPU clock" represents the total time spent on the CPU and "task clock" indicates the time spent by the profiled task. The feature "co.task.clock" refers to the task clock of the co-running task while the feature "task.clock" refers to the task clock of the profiled task itself. The feature "task clock" is very important because it can tell how much percentage of CPU the task has got. In addition, the "context-switches", "CPU-cycles" (i.e. CPU frequency), "bus cycles" and "ref cycles" are important as well because they can reflect the degree of the resource interference of the time-sharing tasks.

5.5 Summary

In this chapter, we reveal the problem of inaccurate estimation of the execution time (due to the resource contention) in the scheduling models. We find out and investigate the influential factors that affect the performance of the co-running tasks. We conduct performance models that consider three scenarios and two task types. The three scenarios are: 1) the tasks are running simultaneously (co-running) on multiple CPU cores in a multi-core processor; 2) the CPU frequency that the task executing with is varied; 3) the tasks are running in a time-sharing manner within the same core. The two task types are: repetitive tasks and new tasks. Several machine learning methods are applied to predict the performance impact of the co-running tasks. Experiments conducted with SPEC 2006 benchmark suite show that our prediction model of performance impact achieves good accuracy on both repetitive tasks and new tasks. In addition, we give detailed analysis of the evaluation results.

CHAPTER 6

Conclusions and Future Work

The work described in this thesis has been concerned with improvement to the task scheduling strategies in multi-core computers. Key contributions are summarized in the first three sections of this chapter. Further work is discussed in Section 6.4.

6.1 Power-aware Scheduling Mechanisms for Virtualized Environments

Cloud computing emerges as one of the most important technologies for interconnecting people and building the so-called Internet of People (IoP). Nowadays, energy consumption in such a system is a critical metric to measure the sustainability and eco-friendliness of the system. Chapter 3 reveals that the traditional scheduling strategies in virtualized systems managed by Xen may lead to performance loss and energy waste, due to the limitation of Xen in adjusting CPU frequency. Four scheduling strategies are presented to remedy this situation, which are the Least performance Loss Scheduling (LLS) strategy, the No performance Loss Scheduling (NLS) strategy, the Best Frequency Match strategy for a single core (BFMS) and the Best Frequency Match strategy for multiple cores (BFMM). These power-aware strategies are developed by identifying the limitation of Xen in scaling the CPU frequency and aim to reduce the energy waste without sacrificing the jobs running performance in the computing systems virtualized by Xen. Least performance Loss Scheduling works by rearranging the execution order of the virtual machines (VMs). No performance Loss Scheduling works by setting a proper initial CPU frequency for running the VMs. Best Frequency Match reduces energy waste and performance loss by allowing the VMs to jump the queue so that the VM that is put into execution best matches the current CPU frequency. Scheduling for both single core and multi-core processors is considered in this chapter. These strategies make use of the scheduling behaviour in the Xen hypervisor and aim to reduce energy consumption while mitigating performance loss. The effectiveness of these strategies is theoretically proved and also evaluated by the experiments.

6.2 Scheduling DAG Applications for Time Sharing Systems

Accurately modeling the makespan of a task is also important for task scheduling strategies. In order to satisfy each task's time constraint (i.e. deadline), the worst-case execution time of a task is taken into consideration when making scheduling decisions. In Chapter 4, we reveal the problem in task makespan modeling and propose the method to model and formulate the makespan with the time-sharing execution. Based on the makespan model, a Task Migration Algorithm and a Task Allocation algorithm are developed, aiming to reduce the actual makespan of the DAG schedule when the DAG is executed in time-sharing in reality. We conduct the extensive experiments with the real-world workflows. The experimental results show that there exists the gap between the makespan in sequential execution, the makespan in time-sharing execution and the makespan obtained by our DAG scheduling algorithm designed for time-sharing systems.

6.3 Contention-aware Prediction for Performance Impact of Task Co-running in Multi-core Computers

Resource interference is another influential factor to the performance of co-running tasks in multi-core computers. In the task scheduling, it is often assumed that the scheduler knows the execution time of the tasks, based on the assumption that the techniques are present to predict the performance of tasks. Chapter 5 investigates the influential factors that affect the performance of the co-running tasks. A performance model is built and several machine learning methods are applied to predict the performance impact of the co-running tasks. Here, we investigate the performance impact of the co-running tasks in three scenarios: 1) the tasks are running simultaneously (co-running) on multiple CPU cores in a multi-core processor; 2) the CPU frequency that the task executing with is varied; 3) the tasks are running in a time-sharing manner within the same core. Experiments conducted with SPEC 2006 benchmark suite show that our prediction model of performance impact achieves high accuracy on both repetitive tasks and new tasks.

6.4 Directions for Future Work

Chapter 3 proposed three power-aware scheduling strategies for virtualized systems managed by Xen. The philosophy used in BFM for reducing performance loss and energy consumption can also be applied to other popular schedulers in Xen, such as Credit, Credit 2, RTDS and ARINC 653 schedulers. In future, we plan to adapt these schedulers to mitigate the performance loss and energy waste. In addition, we will optimize the boost policy of the Credit scheduler to improve the performance of I/O-intensive tasks. Furthermore, we plan to improve the throughout, the execution efficiency and the energy consumption of the DAG and the parallel jobs. Moreover, we will try to tackle the task scheduling in multi-core computers by taking into account load balance and migration cost. We also plan to apply the game theory to investigate the trade-off between task makespan and energy consumption.

Chapter 4 investigated the impact of the time-sharing execution on the DAG makespan, and proposed the method to model the makespan under the time-sharing execution. In future, we plan to extend our research in three folds: 1) constructing an energy consumption model for DAG under the time-sharing execution; 2) developing the DAG scheduling algorithms for the time-sharing execution and taking both makespan and energy consumption into account; 3) adapting the proposed method to the scenario of mapping a single DAG application to multi-core processors, which will be further extended to map multiple DAG applications to multi-core processors. In this more complex scenario, a new makespan model will be constructed. Based on the new makespan model for multiple DAG applications, a task scheduling algorithm will be proposed to minimize the overall energy consumption while satisfying the DAGs' deadline constraints.

Chapter 5 investigated the influential factors that impact on the performance when the tasks are co-running on multi-core computers and developed the machine learning-based prediction frameworks to predict the performance of the co-running tasks. In future, our research will be extended to a more complex scenario: modelling the performance impact when DAG applications run on multi-core processors. Then, we will apply the deep learning technique to further improve the prediction accuracy. Furthermore, distributed training models can be developed to improve the efficiency and reduce the response time needed for prediction.

In addition to the execution time, the energy consumption also acts as a key role in scheduling strategies. Another research direction following this work is to develop a prediction framework that can estimate the temperature induced by the co-running tasks. Different combination of the execution tasks may lead to different thermal stress induced on hardware. The architecture of the processing node is an important attribute of the temperature prediction. Our study shows that the same workload executed on the processing nodes with different architectures may result in different temperature, which will further lead to different energy consumption. The relationship between temperature and executing frequency is non-linear. Thus, accurate prediction for the executing temperature of the co-running tasks under a specific frequency can help reduce the energy consumption of a single task. It will also help make better decisions for allocating tasks to the computing nodes in a heterogeneous cluster, reducing the overall energy consumption of the cluster.

Bibliography

[1] Enhanced intel speedstep technology for the intel pentium m processor. *White Paper*.

[2] J. Ahn, C. H. Park, and J. Huh. Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 394–405. IEEE, 2014.

[3] M. M. Alves and L. M. de Assumpção Drummond. A multivariate and quantitative model for predicting cross-application interference in virtual environments. *Journal of Systems and Software*, 128:150–163, 2017.

[4] K. M. Attia, M. A. El-Hosseini, and H. A. Ali. Dynamic power management techniques in multi-core architectures: A survey study. *Ain Shams Engineering Journal*, 8(3):445 – 456, 2017. ISSN 2090-4479. doi: https://doi.org/10.1016/j.asej.2015.08.010. URL http://www.sciencedirect.com/science/article/pii/S2090447915001380.

[5] W. Bao, S. Tavarageri, F. Ozguner, and P. Sadayappan. Pwcet: Power-aware worst case execution time analysis. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 439–447, Sept 2014. doi: 10.1109/ICPPW.2014.64.

[6] A. Beloglazov, J. Abawajy, and R. Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems*, 28(5):755 – 768, 2012. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2011.04.017. URL http://www.sciencedirect.com/science/article/pii/S0167739X11000689. Special Section: Energy efficiency in large-scale distributed systems.

[7] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin.

[8] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001. ISSN 1573-0565. doi: 10.1023/A:1010933404324. URL https://doi.org/10.1023/A:1010933404324.

[9] L. C. Canon and L. Philippe. On the heterogeneity bias of cost matrices for assessing scheduling algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 28(6):1675–1688, June 2017. ISSN 1045-9219. doi: 10.1109/TPDS.2016.2629503.

[10] A. Castiglione, M. Gribaudo, M. Iacono, and F. Palmieri. Modeling performances of concurrent big data applications. *Software: Practice and Experience*, 45(8):1127–1144. doi: 10.1002/spe.2269. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2269.

[11] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 340–351. IEEE, 2005.

[12] J. Chen, K. Li, Z. Tang, K. Bilal, S. Yu, C. Weng, and K. Li. A parallel random forest algorithm for big data in a spark cloud computing environment. *IEEE Transactions on Parallel and Distributed Systems*, 28(4): 919–933, April 2017. ISSN 1045-9219. doi: 10.1109/TPDS.2016.2603511.

[13] W. Chen, G. Xie, R. Li, Y. Bai, C. Fan, and K. Li. Efficient task scheduling for budget constrained parallel applications on heterogeneous

cloud computing systems. *Future Generation Computer Systems*, 74:1 – 11, 2017. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2017.03.008. URL http://www.sciencedirect.com/science/article/pii/S0167739X16304411.

[14] Y. Cheng, W. Chen, Z. Wang, and Y. Xiang. Precise contention-aware performance prediction on virtualized multicore system. *Journal of Systems Architecture*, 72:42 – 50, 2017. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2016.06.006. URL http://www.sciencedirect.com/science/article/pii/S1383762116300649. Design Automation for Embedded Ubiquitous Computing Systems.

[15] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Performance Evaluation Review*, 35(2): 42–51, 2007.

[16] K. Chronaki, A. Rico, M. Casas, M. Moret, R. M. Badia, E. Ayguad, J. Labarta, and M. Valero. Task scheduling techniques for asymmetric multi-core systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):2074–2087, July 2017. ISSN 1045-9219. doi: 10.1109/TPDS.2016.2633347.

[17] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20 (3):273–297, Sep 1995. ISSN 1573-0565. doi: 10.1007/BF00994018. URL https://doi.org/10.1007/BF00994018.

[18] R. F. da Silva, G. Juve, M. Rynge, E. Deelman, and M. Livny. Online task resource consumption prediction for scientific workflows. *Parallel Processing Letters*, 25(03):1541003, 2015. doi: 10.1142/S0129626415410030. URL https://doi.org/10.1142/S0129626415410030.

[19] D. Dauwe, E. Jonardi, R. Friese, S. Pasricha, A. A. Maciejewski, D. A. Bader, and H. J. Siegel. A methodology for co-location aware application performance modeling in multicore computing. In *Parallel and Distributed*

Processing Symposium Workshop (IPDPSW), 2015 IEEE International, pages 434–443. IEEE, 2015.

[20] D. Dauwe, E. Jonardi, R. D. Friese, S. Pasricha, A. A. Maciejewski, D. A. Bader, and H. J. Siegel. Hpc node performance and energy modeling with the co-location of applications. *The Journal of Supercomputing*, 72(12): 4771–4809, 2016.

[21] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger. Pegasus: a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015. doi: 10.1016/j.future.2014.10.008. URL http://pegasus.isi.edu/publications/2014/2014-fgcs-deelman.pdf. Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575.

[22] G. Dhiman, G. Marchetti, and T. Rosing. vgreen: a system for energy efficient computing in virtualized environments. In *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*, pages 243–248. ACM, 2009.

[23] B. Dorronsoro, S. Nesmachnow, J. Taheri, A. Y. Zomaya, E.-G. Talbi, and P. Bouvry. A hierarchical approach for energy-efficient scheduling of large workloads in multicore distributed systems. *Sustainable Computing: Informatics and Systems*, 4(4):252 – 261, 2014. ISSN 2210-5379. doi: https://doi.org/10.1016/j.suscom.2014.08.003. URL http://www.sciencedirect.com/science/article/pii/S2210537914000444. Special Issue on Energy Aware Resource Management and Scheduling (EARMS).

[24] P. J. Drongowski. Perf tutorial: Counting hardware performance events, 2015. URL http://sandsoftwaresound.net/perf/perf-tut-count-hw-events/.

[25] M. Duan, K. Li, X. Liao, and K. Li. A parallel multiclassification algorithm for big data using an extreme learning machine. *IEEE Transactions on Neural Networks and Learning Systems*, 29(6):2337–2351, June 2018. ISSN 2162-237X.

[26] M. Frank, M. Hilbrich, S. Lehrig, and S. Becker. Parallelization, modeling, and performance prediction in the multi-/many core area: A systematic literature review. In *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, pages 48–55. IEEE, 2017.

[27] S. Fu, L. He, C. Huang, X. Liao, and K. Li. Performance optimization for managing massive numbers of small files in distributed file systems. 2014.

[28] M. R. Ga-rey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA, 1979.

[29] A. Girault, C. Prvot, S. Quinton, R. Henia, and N. Sordon. Improving and estimating the precision of bounds on the worst-case latency of task chains. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2018. ISSN 0278-0070. doi: 10.1109/TCAD.2018.2861016.

[30] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. Technical report, May 2011.

[31] X. Guoqi, X. Xiongren, L. Renfa, and L. Keqin. Schedule length minimization of parallel applications with energy consumption constraints using heuristics on heterogeneous distributed systems. *Concurrency and Computation: Practice and Experience*, 29(16):e4024. doi: 10.1002/cpe.4024. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4024. e4024 cpe.4024.

[32] S. Hahn, J. Reineke, and R. Wilhelm. Towards compositionality in execution time analysis: Definition and challenges. *SIGBED Rev.*, 12(1):

28–36, Mar. 2015. ISSN 1551-3688. doi: 10.1145/2752801.2752805. URL http://doi.acm.org/10.1145/2752801.2752805.

[33] J. Han, S. Jeon, Y.-r. Choi, and J. Huh. Interference management for distributed parallel applications in consolidated clusters. *SIGPLAN Not.*, 51 (4):443–456, Mar. 2016. ISSN 0362-1340. doi: 10.1145/2954679.2872388. URL http://doi.acm.org/10.1145/2954679.2872388.

[34] D. Hardy and I. Puaut. Static probabilistic worst case execution time estimation for architectures with faulty instruction caches. *Real-Time Systems*, 51(2):128–152, Mar 2015. ISSN 1573-1383. doi: 10.1007/s11241-014-9212-x. URL https://doi.org/10.1007/s11241-014-9212-x.

[35] L. He, S. A. Jarvis, D. P. Spooner, X. Chen, and G. R. Nudd. Dynamic scheduling of parallel jobs with qos demands in multiclusters and grids. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 402–409. IEEE Computer Society, 2004.

[36] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[37] Y. Hu, C. Liu, K. Li, X. Chen, and K. Li. Slack allocation algorithm for energy minimization in cluster systems. *Future Generation Computer Systems*, 74:119 – 131, 2017. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2016.08.022. URL http://www.sciencedirect.com/science/article/pii/S0167739X16302941.

[38] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682 – 692, 2013. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2012.08.015. URL http://www.sciencedirect.com/science/article/pii/S0167739X12001732. Special Section: Recent Developments in High Performance Computing and Security.

[39] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 485–497, Santa Clara, CA, 2015. USENIX Association. ISBN 978-1-931971-225. URL https://www.usenix.org/conference/atc15/technical-session/presentation/karanasos.

[40] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for i/o performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 101–110. ACM, 2009.

[41] N. Kim, J. Cho, and E. Seo. Energy-credit scheduler: An energy-aware virtual machine scheduler for cloud systems. *Future Generation Computer Systems*, 32(0):128 – 137, 2014. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2012.05.019. URL http://www.sciencedirect.com/science/article/pii/S0167739X1200115X. Special Section: The Management of Cloud Systems, Special Section: Cyber-Physical Society and Special Section: Special Issue on Exploiting Semantic Technologies with Particularization on Linked Data over Grid and Cloud Architectures.

[42] W. Kuang, L. E. Brown, and Z. Wang. Modeling cross-architecture co-tenancy performance interference. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 231–240, May 2015. doi: 10.1109/CCGrid.2015.152.

[43] D. Li and J. Wu. Energy-aware scheduling for aperiodic tasks on multi-core processors. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 361–370. IEEE, 2014.

[44] D. Li, C. Chen, J. Guan, Y. Zhang, J. Zhu, and R. Yu. Dcloud: Deadline-aware resource allocation for cloud computing jobs. *IEEE Transactions*

on Parallel and Distributed Systems, 27(8):2248–2260, Aug 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2489646.

[45] H.-W. Li, Y.-S. Wu, Y.-Y. Chen, C.-M. Wang, and Y.-N. Huang. Application execution time prediction for effective cpu provisioning in virtualization environment. *IEEE Transactions on Parallel and Distributed Systems*, 28(11):3074–3088, 2017.

[46] K. Li, X. Tang, and K. Li. Energy-efficient stochastic task scheduling on heterogeneous computing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 25(11):2867–2876, Nov 2014. ISSN 1045-9219. doi: 10.1109/TPDS.2013.270.

[47] L. Li, Z. Miao, B. Rajkumar, and F. Qi. Deadline-constrained coevolutionary genetic algorithm for scientific workflow scheduling in cloud computing. *Concurrency and Computation: Practice and Experience*, 29(5): e3942. doi: 10.1002/cpe.3942. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3942. e3942 CPE-16-0064.R2.

[48] X. Li, Y. Zhao, Y. Li, L. Ju, and Z. Jia. An improved energy-efficient scheduling for precedence constrained tasks in multiprocessor clusters. In X.-h. Sun, W. Qu, I. Stojmenovic, W. Zhou, Z. Li, H. Guo, G. Min, T. Yang, Y. Wu, and L. Liu, editors, *Algorithms and Architectures for Parallel Processing*, pages 323–337, Cham, 2014. Springer International Publishing.

[49] Y. Li, J. Niu, M. Atiquzzaman, and X. Long. Energy-aware scheduling on heterogeneous multi-core systems with guaranteed probability. *Journal of Parallel and Distributed Computing*, 103:64 – 76, 2017. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2016.11.014. URL http://www.sciencedirect.com/science/article/pii/S0743731516301745. Special Issue on Scalable Cyber-Physical Systems.

[50] Q. Liao, S. Jiang, Q. Hei, T. Li, and Y. Yang. Scheduling stochastic tasks

with precedence constrain on cluster systems with heterogenous communication architecture. In G. Wang, A. Zomaya, G. Martinez, and K. Li, editors, *Algorithms and Architectures for Parallel Processing*, pages 85–99. Springer International Publishing, 2015.

[51] C.-C. Lin, C.-J. Chang, Y.-C. Syu, J.-J. Wu, P. Liu, P.-W. Cheng, and W.-T. Hsu. An energy-efficient task scheduler for multi-core platforms with per-core dvfs based on task characteristics. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 381–390. IEEE, 2014.

[52] C.-C. Lin, Y.-C. Syu, C.-J. Chang, J.-J. Wu, P. Liu, P.-W. Cheng, and W.-T. Hsu. Energy-efficient task scheduling for multi-core platforms with per-core dvfs. *Journal of Parallel and Distributed Computing*, 86:71 – 81, 2015. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2015.08.004. URL http://www.sciencedirect.com/science/article/pii/S0743731515001422.

[53] C. Liu, K. Li, C. Xu, and K. Li. Strategy configurations of multiple users competition for cloud service reservation. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):508–520, 2016.

[54] M. Liu, C. Li, and T. Li. Understanding the impact of vcpu scheduling on dvfs-based power management in virtualized cloud environment. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pages 295–304. IEEE, 2014.

[55] K. Maheshwari, E.-S. Jung, J. Meng, V. Morozov, V. Vishwanath, and R. Kettimuthu. Workflow performance improvement using model-based scheduling over multiple clusters and clouds. *Future Gener. Comput. Syst.*, 54(C):206–218, Jan. 2016. ISSN 0167-739X. doi: 10.1016/j.future.2015.03.017. URL https://doi.org/10.1016/j.future.2015.03.017.

[56] V. S. Marco, B. Taylor, B. Porter, and Z. Wang. Improving spark application throughput via memory aware task co-location: A mixture of experts approach. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 95–108, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4720-4. doi: 10.1145/3135974.3135984. URL http://doi.acm.org/10.1145/3135974.3135984.

[57] A. Matsunaga and J. A. Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504. IEEE Computer Society, 2010.

[58] J. Mei, K. Li, and K. Li. Customer-satisfaction-aware optimal multiserver configuration for profit maximization in cloud computing. *IEEE Transactions on Sustainable Computing*, 2(1):17–29, Jan 2017. ISSN 2377-3782. doi: 10.1109/TSUSC.2017.2667706.

[59] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 211–221, July 2015. doi: 10.1109/ECRTS.2015.26.

[60] I. A. Moschakis and H. D. Karatza. A meta-heuristic optimization approach to the scheduling of bag-of-tasks applications on heterogeneous clouds with multi-level arrivals and critical jobs. *Simulation Modelling Practice and Theory*, 57:1 – 25, 2015. ISSN 1569-190X. doi: https://doi.org/10.1016/j.simpat.2015.04.009. URL http://www.sciencedirect.com/science/article/pii/S1569190X15000787.

[61] T. Mudge. Power: a first-class architectural design constraint. *Computer*, 34(4):52–58, Apr 2001. ISSN 0018-9162. doi: 10.1109/2.917539.

[62] E. E. N., K. Michael, and R. Ramakrishnan. Energy-efficient server clusters. In *Proceedings of the 2Nd International Conference on Power-*

aware Computer Systems, PACS'02, pages 179–197, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-01028-9. URL http://dl.acm.org/citation.cfm?id=1766991.1767007.

[63] S. Nesmachnow, S. Iturriaga, B. Dorronsoro, and A. Tchernykh. Multiobjective energy-aware workflow scheduling in distributed datacenters. In I. Gitler and J. Klapp, editors, *High Performance Computer Applications*, pages 79–93, Cham, 2016. Springer International Publishing. ISBN 978-3-319-32243-8.

[64] J. Nowotsch. *Interference-sensitive Worst-case Execution Time Analysis for Multi-core Processors*. doctoralthesis, Universität Augsburg, 2014.

[65] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522716. URL http://doi.acm.org/10.1145/2517349.2522716.

[66] M. A. Oxley, S. Pasricha, A. A. Maciejewski, H. J. Siegel, J. Apodaca, D. Young, L. Briceno, J. Smith, S. Bahirat, B. Khemka, A. Ramirez, and Y. Zou. Makespan and energy robust stochastic static resource allocation of a bag-of-tasks to a heterogeneous computing system. *IEEE Transactions on Parallel and Distributed Systems*, 26(10):2791–2805, Oct. 2015. ISSN 1045-9219. doi: 10.1109/TPDS.2014.2362921. URL doi.ieeecomputersociety.org/10.1109/TPDS.2014.2362921.

[67] M. A. Oxley, E. Jonardi, S. Pasricha, A. A. Maciejewski, H. J. Siegel, P. J. Burns, and G. A. Koenig. Rate-based thermal, power, and co-location aware resource management for heterogeneous data centers. *Journal of Parallel and Distributed Computing*, 112:126–139, 2018.

[68] B. Paul, D. Boris, F. Keir, H. Steven, H. Tim, H. Alex, N. Rolf, P. Ian, and

W. Andrew. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, Oct. 2003. ISSN 0163-5980. doi: 10.1145/1165389.945462. URL http://doi.acm.org/10.1145/1165389.945462.

[69] T. P. Pham, J. J. Durillo, and T. Fahringer. Predicting workflow task execution time in the cloud using a two-stage machine learning approach. *IEEE Transactions on Cloud Computing*, pages 1–1, 2018. ISSN 2168-7161. doi: 10.1109/TCC.2017.2732344.

[70] J.-M. Pierson and H. Casanova. On the utility of dvfs for power-aware job placement in clusters. In *Euro-Par 2011 Parallel Processing*, pages 255–266. Springer, 2011.

[71] F. Ramezani, J. Lu, J. Taheri, and F. K. Hussain. Evolutionary algorithm-based multi-objective task scheduling optimization model in cloud environments. *World Wide Web*, 18(6):1737–1757, Nov 2015. ISSN 1573-1413. doi: 10.1007/s11280-015-0335-3. URL https://doi.org/10.1007/s11280-015-0335-3.

[72] F. Ramezani, J. Lu, J. Taheri, and A. Y. Zomaya. A multi-objective load balancing system for cloud environments. *The Computer Journal*, 60(9): 1316–1337, 2017. doi: 10.1093/comjnl/bxw109. URL http://dx.doi.org/10.1093/comjnl/bxw109.

[73] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 36:1–36:15, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901354. URL http://doi.acm.org/10.1145/2901318.2901354.

[74] J. Reineke and R. Wilhelm. Impact of resource sharing on performance and performance prediction. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 96:1–96:2, 3001 Leuven,

Belgium, Belgium, 2014. European Design and Automation Association. ISBN 978-3-9815370-2-4. URL http://dl.acm.org/citation.cfm?id=2616606.2616724.

[75] J. Sahuquillo, H. Hassan, S. Petit, J. L. March, and J. Duato. A dynamic execution time estimation model to save energy in heterogeneous multicores running periodic tasks. *Future Generation Computer Systems*, 56: 211 – 219, 2016. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2015.06.011. URL http://www.sciencedirect.com/science/article/pii/S0167739X15002216.

[76] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49 (4):404–435, Jul 2013. ISSN 1573-1383. doi: 10.1007/s11241-012-9166-9. URL https://doi.org/10.1007/s11241-012-9166-9.

[77] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-efficient sampling for performance prediction of configurable systems (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 342–352, Nov 2015. doi: 10.1109/ASE.2015.45.

[78] A. Shahid, M. Fahad, R. R. Manumachu, and A. Lastovetsky. Additivity: A selection criterion for performance events for reliable energy predictive modeling. *Supercomputing Frontiers and Innovations*, 4:50–65, 12/2017 2017. doi: 10.14529/jsfi170404.

[79] M. Shanmugasundaram, R. Kumar, D. Shinde, and H. M. Kittur. Comparative analysis of scientific workflow scheduling in cloud environment. In *2017 Innovations in Power and Advanced Computing Technologies (i-PACT)*, pages 1–7, April 2017. doi: 10.1109/IPACT.2017.8245020.

[80] G. L. Stavrinides and H. D. Karatza. Scheduling techniques for complex workloads in distributed systems. In *Proceedings of the 2Nd International Conference on Future Networks and Distributed Systems*, ICFNDS '18,

pages 34:1–34:6, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6428-7. doi: 10.1145/3231053.3231087. URL http://doi.acm.org/10.1145/3231053.3231087.

[81] X. Tang, X. Liao, J. Zheng, and X. Yang. Energy efficient job scheduling with workload prediction on cloud data center. *Cluster Computing*, Feb 2018. ISSN 1573-7543. doi: 10.1007/s10586-018-2154-7. URL https://doi.org/10.1007/s10586-018-2154-7.

[82] Z. Tang, W. Ma, K. Li, and K. Li. A data skew oriented reduce placement algorithm based on sampling. *IEEE Transactions on Cloud Computing*, 2016.

[83] Z. Tang, L. Qi, Z. Cheng, K. Li, S. U. Khan, and K. Li. An energy-efficient task scheduling algorithm in dvfs-enabled cloud environment. *Journal of Grid Computing*, 14(1):55–74, 2016.

[84] G. Terzopoulos and H. D. Karatza. Bag-of-task scheduling on power-aware clusters using a dvfs-based mechanism. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 833–840, May 2014. doi: 10.1109/IPDPSW.2014.95.

[85] G. Terzopoulos and H. D. Karatza. Bag-of-tasks load balancing on power-aware clusters. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 135–142, Feb 2016. doi: 10.1109/PDP.2016.25.

[86] J. Tu, T. Yang, Y. Zhang, and J. Sun. Particle swarm optimization based task scheduling for multi-core systems under aging effect. In *2017 International Conference on Progress in Informatics and Computing (PIC)*, pages 271–276, Dec 2017. doi: 10.1109/PIC.2017.8359556.

[87] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino,

O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. doi: 10.1145/2523616.2523633. URL http://doi.acm.org/10.1145/2523616.2523633.

[88] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 363–378, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-29-4. URL http://dl.acm.org/citation.cfm?id=2930611.2930635.

[89] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741964. URL http://doi.acm.org/10.1145/2741948.2741964.

[90] G. von Laszewski, L. Wang, A. J. Younge, and X. He. Power-aware scheduling of virtual machines in dvfs-enabled clusters. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, Aug 2009. doi: 10.1109/CLUSTR.2009.5289182.

[91] G. Von Laszewski, L. Wang, A. J. Younge, and X. He. Power-aware scheduling of virtual machines in dvfs-enabled clusters. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[92] L. Wang and Y. Lu. Efficient power management of heterogeneous soft real-time clusters. In *Real-Time Systems Symposium, 2008*, pages 323–332, Nov 2008. doi: 10.1109/RTSS.2008.31.

[93] L. Wang, S. U. Khan, D. Chen, J. Koodziej, R. Ranjan, C. zhong Xu, and A. Zomaya. Energy-aware parallel task scheduling in a cluster. *Future Generation Computer Systems*, 29(7):1661 – 1670, 2013. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2013.02.010. URL http://www.sciencedirect.com/science/article/pii/S0167739X13000484. Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services Cloud Computing and Scientific Applications Big Data, Scalable Analytics, and Beyond.

[94] Wikipedia contributors. Linear regression — Wikipedia, the free encyclopedia, 2018. URL https://en.wikipedia.org/w/index.php?title=Linear_regression&oldid=856750108. [Online; accessed 20-September-2018].

[95] Wikipedia contributors. Computer multitasking — Wikipedia, the free encyclopedia, 2018. URL https://en.wikipedia.org/w/index.php?title=Computer_multitasking&oldid=851906963. [Online; accessed 22-September-2018].

[96] Wikipedia contributors. Naive bayes classifier — Wikipedia, the free encyclopedia, 2018. URL https://en.wikipedia.org/w/index.php?title=Naive_Bayes_classifier&oldid=859191310. [Online; accessed 20-September-2018].

[97] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem&mdash;overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. ISSN 1539-9087. doi: 10.1145/1347375.1347389. URL http://doi.acm.org/10.1145/1347375.1347389.

[98] C. Witt, M. Bux, W. Gusew, and U. Leser. Predictive Performance Mod-

eling for Distributed Computing using Black-Box Monitoring and Machine Learning. *ArXiv e-prints*, May 2018.

[99] H. Wu, X. Hua, Z. Li, and S. Ren. Resource and instance hour minimization for deadline constrained dag applications using computer clouds. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):885–899, March 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2411257.

[100] S. Xi, W. J., C. Lu, and G. C. Rt-xen: Towards real-time hypervisor scheduling in xen. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 39–48, Oct 2011.

[101] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu. vslicer: latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 3–14. ACM, 2012.

[102] Y. Xu, K. Li, J. Hu, and K. Li. A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. *Information Sciences*, 270:255 – 287, 2014. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2014.02.122. URL `http://www.sciencedirect.com/science/article/pii/S002002551400228X`.

[103] X. Yao, P. Geng, and X. Du. A task scheduling algorithm for multi-core processors. In *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 259–264, Dec 2013. doi: 10.1109/PDCAT.2013.47.

[104] C. Yiyu, D. Amitayu, Q. Wubi, S. Anand, W. Qian, and G. Natarajan. Managing server energy and operational costs in hosting centers. *SIGMETRICS Perform. Eval. Rev.*, 33(1):303–314, June 2005. ISSN 0163-5999. doi: 10.1145/1071690.1064253. URL `http://doi.acm.org/10.1145/1071690.1064253`.

[105] C. Yu, L. Qin, and J. Zhou. A multicore periodical preemption virtual machine scheduling scheme to improve the performance of computational tasks. *The Journal of Supercomputing*, 67(1):254–276, 2014.

[106] L. Yuan, P. Jia, and Y. Yang. Efficient scheduling of dag tasks on multi-core processor based parallel systems. In *TENCON 2015 - 2015 IEEE Region 10 Conference*, pages 1–6, Nov 2015. doi: 10.1109/TENCON.2015.7373088.

[107] H.-E. Zahaf, A. E. H. Benyamina, R. Olejnik, and G. Lipari. Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms. *Journal of Systems Architecture*, 74:46 – 60, 2017. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2017.01.002. URL http://www.sciencedirect.com/science/article/pii/S138376211730019X.

[108] F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, Sept 2009. ISSN 0018-9340. doi: 10.1109/TC.2009.58.

[109] K. Zhang, A. Guliani, S. Ogrenci-Memik, G. Memik, K. Yoshii, R. Sankaran, and P. Beckman. Machine learning-based temperature prediction for runtime thermal management across system components. *IEEE Transactions on Parallel and Distributed Systems*, 29(2):405–419, Feb 2018. ISSN 1045-9219. doi: 10.1109/TPDS.2017.2732951.

[110] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465388. URL http://doi.acm.org/10.1145/2465351.2465388.

[111] J. Zhao, H. Cui, J. Xue, and X. Feng. Predicting cross-core performance

interference on multicore processors with regression analysis. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1443–1456, May 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2442983.

[112] Y. Zhao, J. Rao, and Q. Yi. Characterizing and optimizing the performance of multithreaded programs under interference. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pages 287–297. IEEE, 2016.

[113] J. Zhou, K. Cao, P. Cong, T. Wei, M. Chen, G. Zhang, J. Yan, and Y. Ma. Reliability and temperature constrained task scheduling for makespan minimization on heterogeneous multi-core platforms. *Journal of Systems and Software*, 133:1 – 16, 2017. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2017.07.032. URL http://www.sciencedirect.com/science/article/pii/S0164121217301607.

[114] H. Zhu, L. He, S. Jarvis, et al. Optimizing job scheduling on multicore computers. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pages 61–70. IEEE, 2014.

[115] Q. Zhu and T. Tung. A performance interference model for managing consolidated workloads in qos-aware clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 170–179. IEEE, 2012.

[116] X. Zhu, C. He, K. Li, and X. Qin. Adaptive energy-efficient scheduling for real-time tasks on dvs-enabled heterogeneous clusters. *Journal of Parallel and Distributed Computing*, 72(6):751 – 763, 2012. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2012.03.005. URL http://www.sciencedirect.com/science/article/pii/S0743731512000706.

[117] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, Dec. 2012. ISSN

0360-0300. doi: 10.1145/2379776.2379780. URL http://doi.acm.org/10.1145/2379776.2379780.