National Technical University of Athens
School of Electrical and Computer Engineering

Computer Science Division
Computing Systems Laboratory

Co-scheduled VM interference in over-subscribed servers

Diploma Thesis
of

Konstantinos Kaffes

Supervisor: Nectarios Koziris
Professor N.T.U.A.

Athens, October 2015

National Technical
University of Athens
School of Electrical and
Computer Engineering
Computer Science Division
Computing Systems Laboratory

Co-scheduled VM interference in over-subscribed servers

DIPLOMA THESIS

OF

Konstantinos Kaffes

Supervisor: Nectarios Koziris
Professor N.T.U.A.

Approved by the committee on the -date-.

....	....	....
Nectarios Koziris	Georgios Goumas	Dimitrios Soudris
Professor N.T.U.A.	Lecturer N.T.U.A.	Associate Professor N.T.U.A.

Athens, October 2015.

....
Konstantinos Kaffes
Electrical and Computer Engineer

© (2015) National Technical University of Athens. All rights reserved.
No part of this thesis may be reproduced, stored in retrieval systems, or transmitted in any form or by any means - eletronic, mechanical, photocopying, or otherwise - for profit or commerial advantage. It may be reprinted, stored or distributed for a non-profit, educational or research purpose, given that its source of origin and this notice are retained. Any questions concerning the use of this thesis for profit or commercial advantage should be addressed to the author. The opinions and conclusions stated in this thesis are expressing the author. They should not be considered as a pronouncement of the National Technical University of Athens.

Περίληψη

Οι πάροχοι υπηρεσιών στο υπολογιστικό νέφος λειτουργούν σε περιβάλλον ανταγωνισμού και πρέπει να ικανοποιήσουν κάθε ανάγκη των χρηστών για υπολογιστικούς πόρους. Παρ΄ όλα αυτά οι περισσότερες υπηρεσίες τους υποχρησιμοποιούνται καθώς οι χρήστες συνεχίζουν να ακολουθούν παραδοσιακές μεθόδους κατανομής των πόρων και ζητάνε παραπάνω πόρους από όσους πραγματικά χρειάζονται. Οι πάροχοι υπερφορτώνουν τα υπολογιστικά τους συστήματα προσπαθώντας να μειώσουν το κόστος και να αυξήσουν την αποδοτικότητα των Datacenters διατηρώντας ταυτόχρονα την ποιότητα των υπηρεσιών που παρέχουν στους χρήστες. Σε αυτή την εργασία παρουσιάζουμε μια νέα μέθοδο χρονοδρομολόγησης που στοχεύει να βελτιώσει την αποδοτικότητα των φυσικών μηχανημάτων διατηρώντας την ποιότητα των υπηρέσιων στις εικονικές μηχανές και λαμβάνοντας υπόψη την επίδραση που ασκεί το ένα πρόγραμμα στην επίδοση του άλλου. Για να αποδείξουμε την αποτελεσματικότητα της προσέγγισής μας παρουσιάζουμε πειραματικά αποτελέσματα για μια ευρεία γκάμα προγραμμάτων που εκτελούνται στο υπολογιστικό νέφος.
Λέξεις-Κλειδιά— αποδοτικότητα πόρων, υπολογιστικό νέφος, χρονοδρομολόγηση, παρακολούθηση

Abstract

Modern Infrastructure-as-a-Service Clouds operate in a competitive environment that caters to any user need for computing resources. However, most cloud services are under-utilized as users moving into the Cloud follow traditional provisioning methods and thus over-provision resources. Cloud operators use over-subscription in an effort to consolidate costs and increase the efficiency of datacenters but such solutions endanger the Quality of Service perceived by the users. In this thesis, we present a novel scheduling approach that aims to improve physical host efficiency while preserving VM QoS by taking into account host oversubscription and the resulting workload interference. To validate our approach, we present experimental results on a wide variety of characteristic user workloads.

Keywords— resource efficiency, cloud computing, scheduling, monitoring

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, κ.Νεκτάριο Κοζύρη, για το ερέθισμα που μου έδωσε να γνωρίσω τον τομέα των υπολογιστικών συστημάτων και τη συνεχή καθοδήγησή του στην ακαδημαϊκή μου πορεία. Θέλω ακόμη να ευχαριστήσω το λέκτορα κ. Γεώργιο Γκούμα και όλα τα μέλη του εργαστηρίου που μου παρείχαν συνεχώς καθοδήγηση και γνώσεις ενώ με περιέβαλαν με εμπιστοσύνη όποτε χρειάστηκε. Ένα ιδιαίτερο ευχαριστώ οφείλω στον υποψήφιο διδάκτορα Βαγγέλη Αγγέλου για την αμέριστη βοήθειά του και για την άριστη συνεργασία που είχαμε κατά τη διάρκεια της εκπόνησης της παρούσας διπλωματικής εργασίας. Τέλος, θέλω να ευχαριστήσω θερμά την οικογένειά μου για τη διαρκή στήριξη της, τους φίλους μου για όσα περάσαμε μαζί και τη Φαίδρα για όλα τα παραπάνω καθώς και για τη συνεχή παρότρυνσή της να προσπαθώ πάντα για το καλύτερο.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

An ever growing amount of computation is done in the cloud. Public cloud providers such as Amazon EC2[2], Windows Azure[3] and Google Compute Engine[4] serve thousands of clients and host tens of thousands of applications every day. There are also private enterprise clouds using technologies like Openstack[5] and VMware vCloud[6]. This type of computing is more flexible and cost effective for users as they can increase the resources they demand on the fly without having to purchase or upgrade hardware. Moreover, operators can drive the costs down by creating large scale Datacenters (DC) and provide shared resources to multiple clients. Many actions have been taken in order to increase the cost efficiency of DCs. Prime examples are the use of commodity hardware and the reduction of the cost of cooling systems. Another relevant method is dynamic voltage scaling in order to reduce the processors' frequency and thus the energy consumption when they are underutilized. However, all these techniques have reached the point of diminishing returns[7, 8] while the cost of upgrading the hardware is huge. Furthermore, energy costs consist a large fraction of the total cost of ownership of the datacenter and the fact that servers are not energy-proportional when their utilization is low contributes to this phenomenon. The server utilization in DCs is very low due to varying workloads, server heterogeneity and Quality-of-Service(QoS) agreements between the provider and the client.

Efforts to co-schedule different jobs in the same server cause interference, a consequential degradation to their performance and required in the past extreme resource reservation. Many solutions have been proposed in order to increase utilization while reducing interference such as exploiting heterogeneity of servers to use energy efficient ones for low demand workloads[9] and awareness of shared resources (last level cache, memory channel, network adapter) between different cores[10]. Delimitrou and Kozyrakis introduced Paragon[11] which combines the above techniques and addresses several open problems. It is a heterogeneity- and interference-aware system which classifies incoming jobs using mini-benchmarks [12], taking into consideration both varying workloads with phases and short-running jobs, and places them to servers trying to avoid violation of QoS agreements.

Leverich and Kozyrakis [13] quantify the impact to latency-sensitive workload performance when such applications operate in a shared cluster environment and are thus co-located with other user workloads. They show that, contrary to the traditional view, some latency-critical workloads can be co-located to improve DC resource utilization, and still achieve good QoS, using a variety of techniques. Lo et al [14] safely co-locate low priority best-effort batch workloads along latency critical services using software and hardware isolation techniques that ensure that the latter's performance does not violate Quality of Service.

Other approaches examined in depth cases of over-subscription. Roytman et al [15] examine both the case of consolidating VMs while keeping performance within some bounds and the case of consolidating VMs as much as possible while trying to minimize performance degradation. In DeepDive [16] Novakovic et al present a system that detects and identifies interference in multiple levels trying to minimize the overhead. Shirinbab and Lundberg [17] identify and quantify performance bottlenecks for VMs when running in oversubscribed virtual environments. Adami et al [18] also explore over-subscription in DCs, particularly when it comes to network resources, as applications' network requirements have become more stringent.

1.2 Contribution

Throughout this work we restrict our attention to the problem of scheduling a set of workloads within a server using as little information as possible. First, we composed a set of diverse workloads running inside Virtual Machines (VMs) that stress different components of a computer system. Then, we executed them in different configurations and obtained data about their resource usage and performance. We obtained the baseline-"optimal" performance of each benchmark by placing each one isolated to a server. We also examined space- and time-sharing characteristics of every pair of workloads. Building on these results we designed and implemented two schedulers which determine the pinning of a given set of workloads with known characteristics that must run on a specific server and compared their performance to the commonly used round robin scheduler. The first scheduler is based on resource usage while the other on expected slowdown. Our approach has a dual goal, in case of underscription to consolidate workloads and save cores and in case of oversubscription to place them in such a way as to improve total performance. The methodology is robust and can be easily adapted to match every system's architecture while only basic and easily extracted characteristics of each workload have to be known in order for the schedulers to operate. We succeeded in both our goals as in our experiments we observed significant savings in core usage as well as a substantial performance increase when consolidation was impossible. To our knowledge this is the first effort which uses time-sharing between workloads while minimizing interference with the goal of achieving further resource efficiency.

Chapter 2

Related Work

2.1 Large Scale Cluster Management

2.1.1 Quasar

Delimitrou and Kozyrakis [19] examined the utilization of a production cluster at Twitter which was managed by Mesos [20]. Mesos is a cluster manager which provides efficient resource isolation and sharing across different frameworks (Hadoop, MPI etc.). It employs fine grained resource sharing improving cluster utilization in comparison to a static partitioning of a cluster, where each partition runs a separate framework. However, despite the use of Mesos the aggregate CPU utilization over the period of a month was found to be less than 20%. Google which uses the more advanced Borg manager achieves utilization between 25% and 30% [21] while cloud facilities that do not co-locate workloads are closer to industry average utilizations of $6\% - 12\%$. This underutilization contributes both to capital and operating expenses due to the energy disproportionality of servers.

Delimitrou and Kozyrakis, identifying that only a small fraction of workloads running on a cluster provide a right-sized reservation, developed Quasar cluster manager. The key features of Quasar are the following:

- *Performance-centric approach:* Instead of asking from the user the low-level resource requirements of the application, it demands performance constraints and then determines the least amount of available resources that are needed to meet these constraints. This approach is more robust as it allows adjustments of the resources allocated to varying workloads, better handling of unknown workloads and simplifies the user's role.

- Using only a small amount of profiling information it employs fast classification techniques to determine the impact of different resource allocations and assignments to application performance.
- *Joint resource allocation and assignment* with the ability of reclassification thus avoiding both overprovisioning for idle workloads and performance degradation for high load ones.

In Paragon [11] Delimitrou and Kozyrakis used collaborative filtering inspired from Netflix Challenge[22] in order to classify workloads regarding interference and heterogeneity. Heterogeneity classification requires profiling runs in two different servers while in interference classification microbenchmarks [12] are injected and the workload's sensitivity to them is quantified. Quasar extends the classification engine of Paragon in two ways. It uses collaborative filtering to estimate the impact of scale-up (more resources allocated to the workload per server) and scale-out (more servers allocated to the workload) on workloads performance and adjusts all classifiers mentioned above according to the application type. The output of the classification phase functions as the input of a greedy scheduler which determines resource allocation and adjustment based on server ranking. Another key aspect of Quasar cluster manager is its on-the-fly phase detection. Workloads are both periodically sampled in place through the injection of microbenchmarks and constantly monitored for Quality of Service violation. In case a phase change is detected, the system deals with it conservatively. It first tries intra-server scale-up or scale-down and if this is not sufficient scale-out or migration is considered. Quasar is evaluated using five scenarios, a single batch job, multiple batch jobs,a low-latency service, stateful latency-critical services and a large scale mixed scenario. In all these scenarios Quasar improved aggregate cluster utilization and individual application performance.

However, Quasar has a few shortcomings. First, even though it makes high quality decisions, it needs information about the full cluster state which might make the decision overhead unacceptable. Furthermore, it requires a significant amount of information about incoming workloads (workload type, QoS constraints, framework) which might not be available in non state-of-the-art clusters.

2.1.2 Tarcil

In order to deal with the low scheduling speed of Quasar Delimitrou et al. developed Tarcil [23] which achieves high-quality and high-speed decisions making it ideal for large clusters. Tarcil uses the workload's resource and sensitivity preferences as they are obtained in Paragon and Quasar. Then, admission control is used in order to determine, in case of high load, whether there are satisfactory resources for the workload. If this is not the case, the workload is queued until appropriate resources are found or a timeout has expired. If the job is admitted Tarcil performs sampling-based scheduling.

In this approach the resources are divided in *Resource Units* which consist of one cpu and the equivalent share of the server's memory, disk and network capacity. The interference and tolerance profile of a workload is expressed as a single number between 0 and 1. If that number is high, the workload is resource intensive and thus requires Resource Units of high quality. Then, the same is done for every Resource Unit in the server and these profiles are normally distributed between 0 and 1. When a scheduling decision needs to be made only a small number of Resource Units are sampled as candidates. The positive aspect of sampling-based scheduling is that the larger the cluster, the better the scheduler performs as in general the distribution of the Resource Units' profiles approximates more closely uniformity. According to the experimental results, Tarcil improves scheduling speed,quality and predictability increasing cluster utilization.

2.1.3 User's Side

There are several different ways with which users can provide their application with resources. They can use reserved,on-demand or hybrid provisioning. In reserved provisioning, servers are reserved for a long period of time. This comes with high upfront cost but compensated by low per-hour cost and predictable performance. In on-demand provisioning servers become available progressively as they become necessary having no upfront cost. However, this comes with high per-hour costs and unpredictable performance as the provider is responsible for determining the need for resources. Finally, hybrid provisioning combines the other two approaches that use both reserved (long-term) and on-demand (short-term) resources. The main challenge in this approach is to decide which resources to use for each job and this selection determines whether this policy is beneficial or not. If implemented correctly it has the potential to combine that best of the two worlds.

Configuration	Cost	Predictability	Flexibility	Usage
Reserved	high upfront low per-hour	high	no	long-term
On-demand	no upfront high per-hour	low	yes	short-term
Hybrid	medium up-front medium per-hour	medium	yes	long-term

Table 2.1: System Configurations

2.2 Other Approaches

In addition to the systems mentioned previously other more specialized methods have been examined with the goal of improving the utilization of datacenters. Lo et al. developed Heracles [14], a feedback based controller that enables the safe colocation of best-effort tasks alongside a latency critical service. Heracles tries to enable aggresive colocation of such workloads using multiple hardware and software isolation mechanisms.

At this point we will present several mechanisms used for resource isolation as they are very useful for achieving better performance in datacenter workloads:

- *cpuset cgroups* provide a mechanism for assigning a set of CPUs to a set of tasks. The dynamic allocation of cores to workloads is necessary in order to achieve core isolation.
- *Cache Isolation Technology* is a hardware mechanism available to recent Intel chips which implements way partitioning of the shared Last Level Cache (LLC). As a result of this, in a highly-associative LLC it is possible to dynamically make fine-grained partitions and dynamically assign them to different jobs.
- *Linux traffic control* is a software mechanism used for network traffic isolation and can set bandwidth limits for workloads running on the server.

Roytman et al also tackle the problem of VM consolidation in PACMan [15], taking into account the average degradation of workloads' performance, mainly due to their interference, vs energy efficiency. Evaluation of their system with SPEC CPU

2006 benchmarks shows that PACMan realizes 30% savings in energy costs and up to 52% reduction in performance degradation compared to consolidation approaches that do not consider degradation. However, given their experimental setup, performance degradation is only considered in terms of CPU utilization, while real-world workloads may suffer degradation due to the sharing of other host's resources. Furthermore, the PACMan scheduler requires an extensive amount of information which is unlikely to be available in real systems.

Kannan et al [24] present a number of prototypes to alleviate co-scheduled VM demands on the shared resources of CMP platforms. By following their suggestions on proper management of sharing the last-level cache among co-scheduled VMs, we can design scheduling algorithms that do not suffer from non-deterministic performance degradation. Although such degradation can happen when workloads are dependent on memory bandwidth, our approach currently focuses on more general-purpose workload co-scheduling.

Apart from generic schedulers in cloud infrastructures, various efforts such as [25], [26] mainly focus on specific types of workloads, e.g., map-reduce jobs. For instance, Omega[26] is a shared-state, optimistic, transaction-based scheduler that appears as an attractive platform for development of specialized schedulers, and illustrates its flexibility by adding a MapReduce scheduler with opportunistic resource adjustment that benefits $50-70\%$ of MapReduce jobs. Although both of these approaches are intriguing, they do not address VM scheduling, take into account only a particular workload type (variable CPU-time map-reduce jobs) and demonstrate that in the over-subscribed case they offer little improvement relative to centralized schedulers.

Finally, Podzimek et al [27] experiment with various CPU-pinning strategies of different VM and LXC-container workloads. They conclude that less common CPU pinning configurations (such as "per-chip" for heavily loaded systems) improve energy efficiency at partial background loads, indicating that systems hosting co-located workloads could benefit from dynamic CPU pinning based on CPU load and workload type. In our work, we observe similar performance and energy efficiency variation, and we adopt an architecture-neutral approach for our CPU-pinning strategy, which we do not claim as optimal, but sufficient and general enough to be applied to a large variety of commodity hardware based DCs.

Chapter 3

Architecture

3.1 Monitoring System

In order to be able to predict the behavior of workloads, consolidate them and ultimately improve their performance a monitoring system is of great importance. The parameters that we use in our analysis and that we need to monitor is the per VM CPU, DiskIO, NetIO and Memory Bandwidth utilization as a percentage of the total resources of the system. Our first approach was to use ganglia[28] a distributed monitoring system primarily used for clusters and grids. It manages to achieve very low per-node overheads and high concurrency while its setup and operation are simple. However it presented a few shortcomings that rendered it useless for our case. First, because of its architecture each VM reports the metrics of interest as it perceives them. Nevertheless, these metrics were not very accurate as they were taken from within the VM and not from the physical host. Second and most important, ganglia cannot measure Memory Bandwidth usage, a critical parameter that we wanted to incorporate into our analysis.

3.1.1 libvirt Statistics

The reasons mentioned above along with the fact that our system is limited to a single server led us to the decision to build our own monitoring system. It is implemented as a daemon written in Python running on the physical host. It uses libvirt [29] in order to gather the CPU mapping and then the CPU, DiskIO and NetIO utilization of the running VMs.

Libvirt is a collection of software which includes a daemon (libvirtd), an API library and a command line utility (virsh). It is used for **VM Management**, **Remote machine support**, **Storage management**, **Network interface management** and **Virtual NAT and Route based networking**. The component that proved to be the most useful in our case was the API library. The daemon uses the library's Python bindings in order to obtain the statistics mentioned above. It associates each VM with the core, the block interface and the network interface it uses and gets the data about its execution on them from the hypervisor, in our case QEMU-KVM.

3.1.2 Perf Statistics

The measurement of the Memory Bandwidth usage is not as simple as that of other resources as hardware support is needed. Every modern processor is equipped with a set of Performance Monitoring Units (PMUs) which are hardware counters that measure microarchitectural events such as clock cycles and cache misses. In our work we use Intel Westemere processors which belong to one of the first families of processors to feature a PMU capable of measuring the per process DRAM requests, an event necessary to our analysis. There exists an official kernel interface, which is called perf_events and provides a high level and generic interface to count and sample hardware and software events. Its main advantage is that users just have to pass the events to measure with the kernel being responsible for the programming of these events onto the correct counters and the management of the PMUs. The tool we use in order to capture the events of our interest is perf [30], a command line tool used to collect performance data from many different sources such as kernel software counters and hardware counters. The important parameter is that it is possible to measure both on system wide and on per process mode. The tool is built on top of the perf_events interface. For the counting of events the perf stat command is used. The occurrences of events are simply aggregated and presented on standard output at the end of an application run or a time period.

In order to calculate the Memory Bandwidth usage we use the following perf events as in [31]:

The consumed Memory Bandwidth of an entire socket is calculated using:

$$consumedMemoryBandwidth =$$

$$\frac{64 \times (UNC\_QMC\_NORMAL\_WRITES + UNC\_QMC\_NORMAL\_READS)}{TimeWindow}$$

Hardware Events	Description
UNC_QMC_NORMAL_READS	Memory Reads
UNC_QMC_NORMAL_WRITES	Memory Writes
OFFCORE_RESPONSE	Requests serviced by DRAM

Table 3.1: Performance Counters

while for the per VM Memory Bandwidth Utilization we use:

$$MemoryBandwidthUtilization = \frac{64 \times OFFCORE\_RESPONSE}{TimeWindow \times totalMemoryBandwidth}$$

The monitoring system stores the per minute average of these metrics in a logfile along with a timestamp.

3.2 Scheduler

The most important component of our system is the scheduler. Scheduling is fundamental to computation and an intrinsic part of the execution system. In our case, the scheduler's job is to place incoming workloads on the cores of a server using only limited information about them. We assume that the set of jobs that must run on the server is predefined and can not change regardless of the overhead they introduce. Interserver workload scheduling is not considered as it is orthogonal to our approach. After the datacenter management system assigns some VMs to run on a server, our scheduler takes over and pins them to available cores according to a policy. We have designed, implemented and evaluated three schedulers. The first is a commonly used round robin scheduler while the other two are based on resource utilization and expected interference of the incoming workloads accordingly and try to find a suitable balance between performance and resource efficiency.

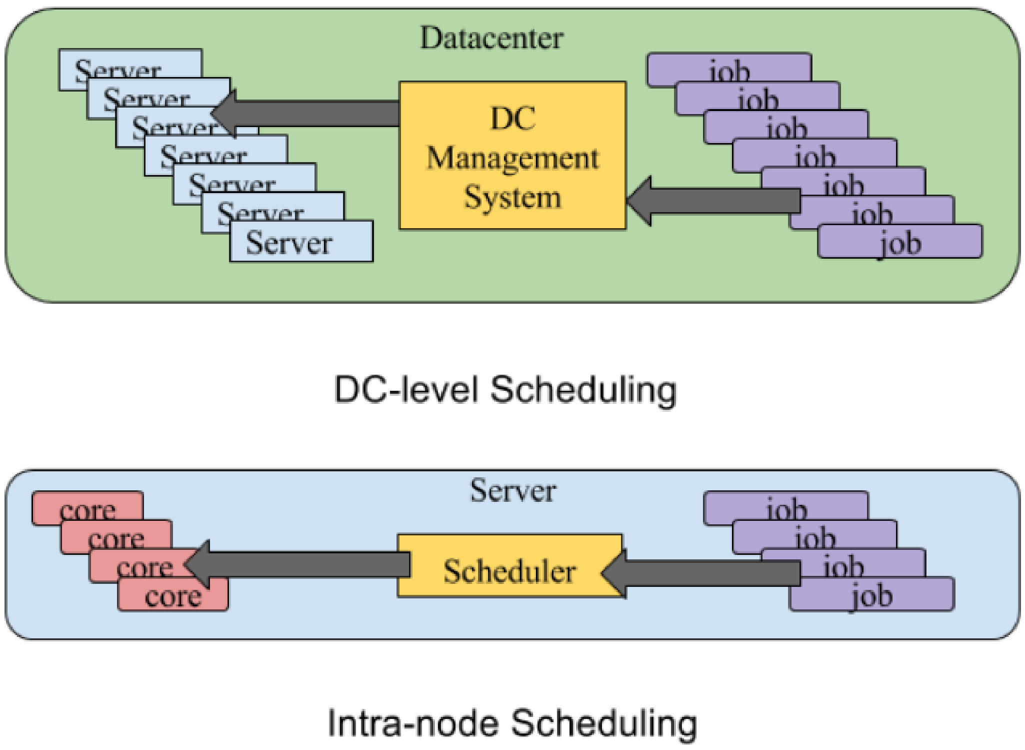


Figure 3.1: System Architecture

The general architecture of the two latter schedulers is described in algorithm [1]. They run as daemons in the background waking up at regular intervals, i.e. case every minute. Every time they wake up they obtain a list of the idle and the running workloads on the server. We consider a workload to be idle if its CPU usage during the last time window was below 2.5%. Using the VirtIO API every idle workload is pinned on a specific CPU of the server and considered to consume zero resources. Then, the running workloads are pinned on the cores of the server according to the implementation of the SelectPinning procedure. Our approach has two discrete but complementary goals. If the server is undersubscribed it tries to consolidate workloads with minimal performance degradation with the goal of saving cores for other jobs or turning them off. In case of oversubscription, it finds a "good" placement that reduces performance degradation induced by workload co-location.

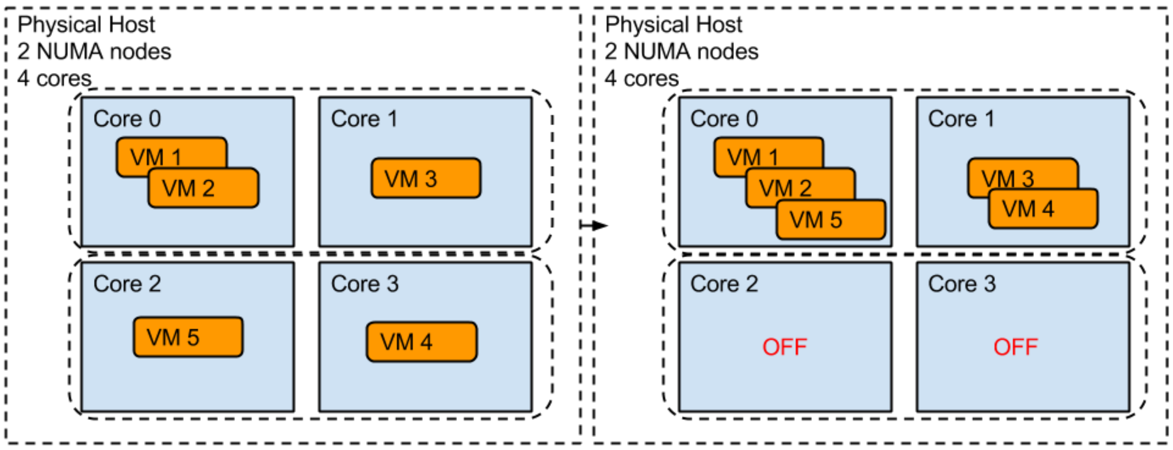


Figure 3.2: VM Consolidation

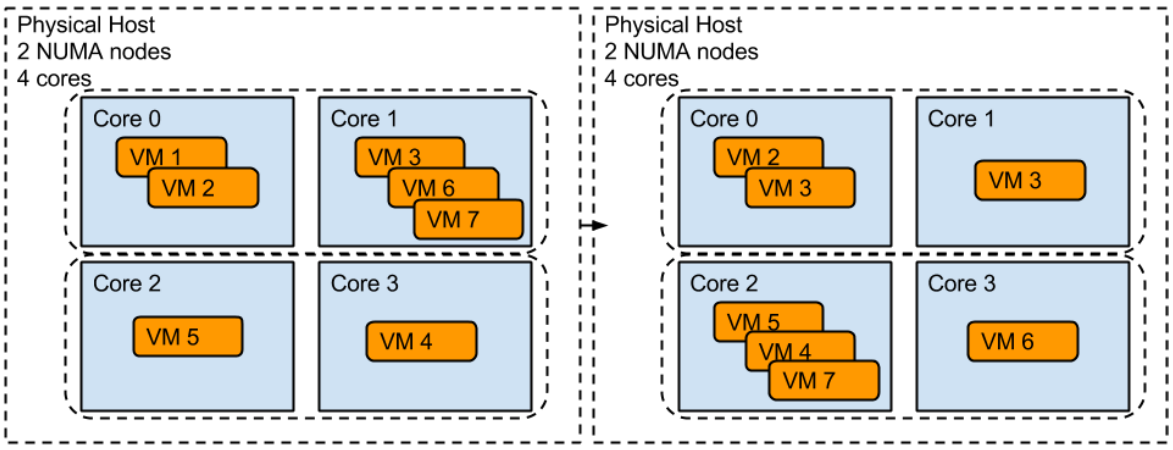


Figure 3.3: VM Rescheduling

3.2.1 Round Robin Scheduler

In our analysis we use as baseline a simple round robin scheduler as described by algorithm [2]. In the beginning of the workload scenario it is given as input the list of workloads. Then it iterates over it and places every workload to a different core and then it starts over. It is interference- and resource-unaware being also unable to detect whether a workload is in running state or idle, since it is agnostic to the monitoring metrics. As a result of this, we assume that throughout the runtime of a scenario all cores hosting a job are considered active even if the VM that hosts the job is idle.

1 Improved Scheduler

```
procedure SCHEDULE:
    while True do
        sleep 60
        idleJobs ← GetIdleJobs()
        runningJobs ← GetRunningJobs()
        for job in idleJobs do
            PlaceVM(0,job)
        end for
        for job in runningJobs do
            targetCPU ← SelectPinning(job)
            PlaceVM(targetCPU,job)
        end for
    end while
end procedure
```

2 Round Robin Scheduler

```
procedure SCHEDULER(JOBS,NCORES):
    targetCore ← 0
    for job in JOBS do
        PlaceVM(targetCore,job)
        targetCore ← (targetCore + 1) % NCORES
    end for
end procedure
```

3.2.2 Resource Based Scheduler (RBS)

The first scheduler is based on the resource utilization of the workloads and is described by algorithm [3]. Through an initial classification phase we obtain the CPU, DiskIO, NetIO and Memory Bandwidth utilization of every workload as percentage of the total resources of the server when running alone on it. If there are N jobs or workload classes those are represented by a $N \times M$ matrix, $\mathbf{U}$, where M : the number of metrics taken into consideration (four in our case). If there is a core, c, with a list A_c of n workloads, $a_1, .., a_n$, in total, we define as load $L(c, A_c)$ the following:

$$L(c, A_c) = \sum_{i=1}^{M} \max(0, \sum_{j=1}^{n} (U[a_j, i]) - \mathit{threshold})$$

We also keep a dictionary of the pinning of all the running jobs that have already been placed to the server. When the decision for the placement of a job, a_{n+1}, is taken, the scheduler checks if there is a core, c, whose load, $L(c, A_c \cup a_{n+1})$, after the new job is added, is zero. If this is the case, the workload's virtual CPU is pinned on that core. If no such core exists, we follow a different approach. The scheduler first determines the load of every core on the server with, $L(c, A_c \cup a_{n+1})$, and without, $L(c, A_c)$, the new workload and then places it to the core whose load will increase the least with the new job. After the placement is made, it updates the resource usage of the system's cores, i.e. the CPU Usage of the selected core, the Memory Bandwidth usage for all cores in the same socket and the NetIO and DiskIO usage for all cores in the server. In our experiments we have selected 120% as threshold not allowing aggressive workload co-location. This process is repeated at every scheduling interval for all VMs but leaves the pinning unchanged if the sets of running and idle jobs remain unchanged too.

3 Resource Utilization Scheduler

```
1: procedure SelectPinning(job):
2:     for i in range(cpus) do
3:         if $L(i, A_i \cup job) = 0$ then
4:             return i
5:         end if
6:     end for
7:     minInter ← $L(0, A_0 \cup job) - L(0, A_0)$
8:     minCPU ← 0
9:     for i in range(1,cpus) do
10:        temp ← $L(i, A_i \cup job) - L(i, A_i)$
11:        if temp<minInter then
12:            minInter ← temp
13:            minCPU ← i
14:        end if
15:    end for
16:    return minCPU
17: end procedure
```

3.2.3 Interference Based Scheduler (IBS)

The second scheduler is based on minimizing the interference between co-located workloads and is described by algorithm [4]. We assume that we have knowledge of the slowdown each job suffers when pinned on the same CPU with every other job, i.e. if there are N jobs or workload classes it is represented by a $N \times N$ matrix, $\mathbf{S}$, containing this information. If there is a workload, a_i, placed in a core, c, with a list A_c of n workloads, $a_1, .., a_n$, in total, we define as interference $I(a_i, c)$ the following:

$$Sum(a_i, c, A_c) = \sum_{\substack{j=1 \\ j \neq i}}^{n} S[a_i, a_j]$$

$$Prod(a_i, c, A_c) = \prod_{\substack{j=1 \\ j \neq i}}^{n} S[a_i, a_j]$$

$$I(a_i, c, A_c) = \frac{Sum(a_i, c) + Prod(a_i, c)}{2}$$

Where $Sum(a_i, c, A_c)$ is the sum of the slowdowns job a_i suffers when placed alone with each other workload that is pinned on core c and $Prod(a_i, c, A_c)$ is the equivalent product.

Moreover, we define the interference of a core, $I(c)$, to be the maximum of the interferences of all the jobs placed in that core.

$$I(c, A_c) = \max_{j=1}^{n} I(a_j, c)$$

When the decision for the placement of a job, a_{n+1}, needs to be taken, the scheduler checks if there exists a core, c whose inteference after the job is placed $I(c, A_c \cup a_{n+1})$ is below a given threshold. If this is the case, the workload's virtual CPU is pinned on that core. If no such core exists, the scheduler first determines the interference of every core on the server with the new workload and then pins it on the core whose interference is minimum after the placement. We have selected 1.5 as the threshold used in this case, creating a rather aggresive scheduler.

In comparison to other approaches[15], where each possible workload set is evaluated live, we do not assume knowledge of the performance degradation suffered by each VM, when consolidated with any set of other VMs. Having just the one-by-one slowdowns available we derived a way to calculate the slowdown a workload suffers when consolidated with multiple other VMs. Our first approach was to use the product of the slowdowns of the workload of interest with all other workloads in the same core as a metric. While this seems reasonable and might be accurate for workloads with large slowdowns, if the one-by-one slowdowns between a set of workloads is small, i.e. $1 - 1.2$, a large number of them could have been placed in the same core without violating the threshold even though their performance could degrade severely. In order to tackle this issue we calculate the average between the product and the sum of the slowdowns of workloads in the same core taking this way also into account the number of the workloads placed in a core.

4 Interference Scheduler

```
 1: procedure SelectPinning(job):
 2:     for i in range(cpus) do
 3:         if $I(i, A_i \cup job) < Threshold$ then
 4:             return i
 5:         end if
 6:     end for
 7:     minInter ← $I(0, A_0 \cup job)$
 8:     minCPU ← 0
 9:     for i in range(1,cpus) do
10:         temp ← $I(i, A_i \cup job)$
11:         if temp<minInter then
12:             minInter ← temp
13:             minCPU ← i
14:         end if
15:     end for
16:     return minCPU
17: end procedure
```

Chapter 4

Experimental Setup

4.1 Setup

Our experimental setup consists of a single server with two Intel Xeon X5650 Processors. The server has twelve 2.66-GHz cores, divided into two six-core sockets that share 12 MB of LLC. The server also features 48 GB of DRAM and one 1-Gb network port. The following summarize the benchmarks used to evaluate the different schedulers. Benchmarks include batch and latency-critical workloads which are large consumers of resources on private and public clouds. A different Virtual Machine (VM) is created for each benchmark. Furthermore, we assume that all VMs have a single virtual core which is pinned to a real core.

4.2 Benchmarks

In our study we have used five different benchmarks in order to capture a wide range of behaviors and potential sources of interference. We stressed all four basic components of a computer system (CPU, NetIO, Memory Bandwidth and DiskIO) to different degrees while we observed its performance.

4.2.1 Blackscholes

Blackscholes[32] was our choice for CPU-heavy workload. It calculates the prices for a portfolio of European options solving analytically the Black-Scholes partial differential equation:

$$\frac{\partial V}{\partial t} + \frac{\sigma^2 S^2}{2} \frac{\partial^2 V}{\partial S^2} + rS \frac{\partial V}{\partial S} - rV = 0$$

This benchmark represents the wide range of PDE solvers that may run on cloud scenarios. The program's performance is limited by the number of floating point operations per second the processor can execute.

4.2.2 Hadoop

Hadoop[33] is a framework for distributing large datasets on computer clusters of commodity hardware. It consists of a storage part (**H**adoop **D**istributed **F**ile **S**ystem) and a processing part (MapReduce). It works by splitting files into blocks and distributing them among nodes in the cluster. In order to process data it transfers code to the nodes and the execution is made parallel.

HDFS uses a master-slave architecture. Each cluster has a single master node (Namenode) which manages the file system namespace and regulates access to files by clients. Moreover, there are Datanodes, usually one per node in the cluster, which manage storage that is directly attached to each node. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

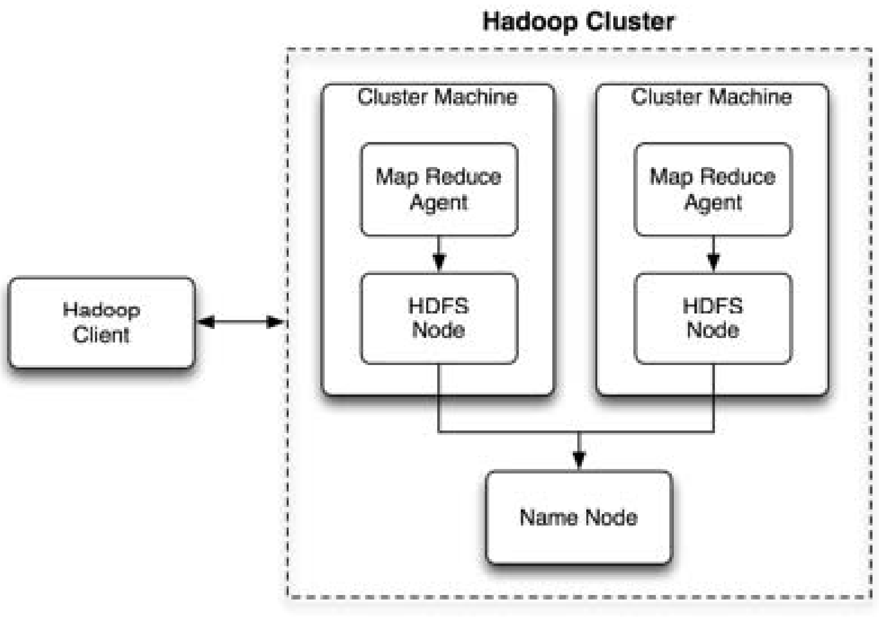


Figure 4.1: Hadoop Architecture [1]

Computations within the Hadoop framework are done using the MapReduce paradigm. The input of a MapReduce program is a set of key/value pairs which produce as an output a different set of key/value pairs. Each MapReduce computation consists of three steps:

- **Map**: Where each worker applies a ”map” function to its local data and writes the output to temporary storage.
- **Shuffle**: Where the results of the previously executed ”map” are redistributed among the nodes based on the output key so that each worker has all the data related to a specific key.
- **Reduce**: Where each worker processes the data related to its output key.

MapReduce offers highly scalable parallel computations which can be executed in commodity hardware as well as fault tolerance. The latter is achieved with the replication of data and the ability of another worker to take over the work of a failed worker node. Moreover, NameNode, which in early versions of Hadoop was considered a single point of failure, has high availability with an active/passive failover.

Hadoop and MapReduce are used in a wide range of applications. Prime examples are distributed sorting, distributed pattern finding, Singular Value Decomposition as well as the complete restructuring of Google's web index.

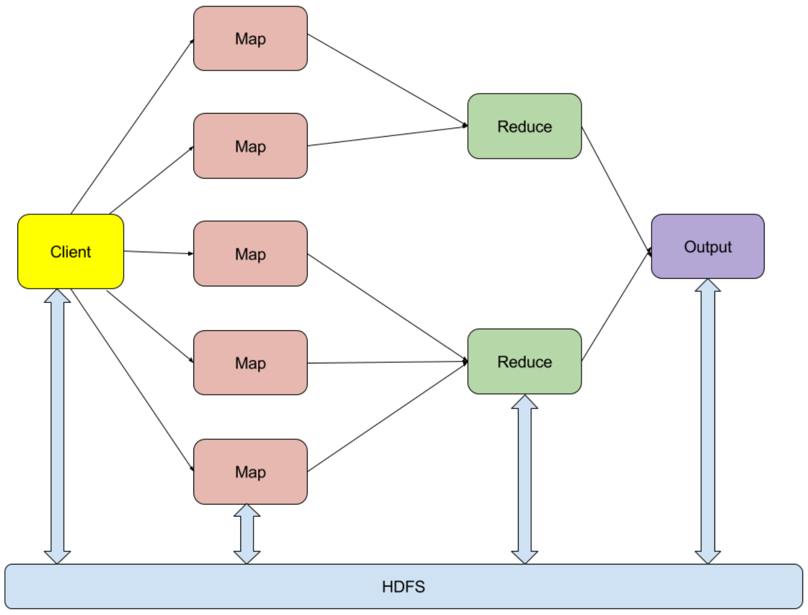


Figure 4.2: MapReduce Paradigm

The Hadoop workload we have chosen is Teragen, a program which generates random data to be used by a subsequent Terasort run and stores them in a file within HDFS. The goal of Terasort is to sort 1TB of data a feat *Yahoo!* has managed to complete within 209 seconds. We preferred Teragen over Terasort due to its continuous and fairly stable DiskIO usage.

4.2.3 Jacobi

A large portion of scientific computing workloads consists of stencil computations. In a stencil computation kernel an array of elements is updated iteratively according to a fixed pattern. Some notable uses of such kernels are computational fluid dynamics, image processing and solving of PDEs. As a representative of this class of workloads the following Jacobi 2D kernel was chosen due to its simplicity and wide usage:

```
for i:=1 to N
    for j:=1 to N
        A[i][j]=0.25(A[i-1][j]+A[i][j-1]
                    +A[i+1][j]+A[i][j+1])
```

Jacobi 2D is characterized by high CPU and memory bus usage, a fact that renders it more sensitive to interference than plain CPU heavy workloads such as blackscholes.

4.2.4 LAMP

LAMP is an acronym originating from the ”Linux, Apache, MySQL and PHP” phrase that refers to a particular software combination which is very popular for web servers. Linux is a free and open source operating system used in the majority of servers. Apache web server supports a wide range of features due to its modular architecture. MySQL is used as LAMP's relational database system while PHP is the main language used for server sided programming.

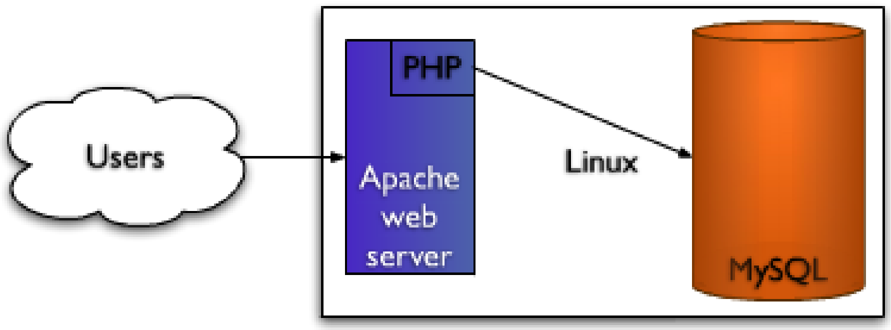


Figure 4.3: LAMP Architecture

The LAMP workload that we used consisted of a group of threads which hit our server with PHP, HTML and REST requests as well as tried to load an image. The main parameter taken into consideration regarding the server's performance in this case is the latency as it is experienced from the clients. We have formulated two variations of the workload, one with a large number of threads and heavy load and a ”lighter” one in order to simulate periods of low traffic in a website. We expect it to be sensitive to interference as it is based on a very thick software stack.

4.2.5 Media Streaming

Media streaming nowadays is one of the most common cloud applications. The creation of Youtube and other similar video hosting services since 2005 and the recent emergence of Netflix, Amazon Instant Video and Hulu are stressing Internet to its limit and need vast server farms in order to retain quality of service. For example, the services mentioned before accounted for 56% of all downstream Internet bandwidth during peak periods in North America for March 2015 according to Sandvine, a Canadian bandwidth-management systems vendor. As a result of this, it is imperative to simulate the behavior of such workloads and try to minimize costs while retaining performance.

We have chosen the Cloudsuite Media Streaming benchmark[34] as the basis of our workload. It consists of two components, a client and a server. The client component emulates real world clients by sending requests to stress a streaming server. Each client requests a number of videos of different length and quality through the RTSP protocol. We substituted Faban driver with bash scripting using xargs command as the method to control the number of threads that simultaneously "hit" the server as the former introduced significant overhead. Darwin Streaming Server is used as the server as it supports RTSP and is relatively lightweight. The significant metric in this case is the server's throughput towards a group of clients and we use it in order to quantify performance degradation. Finally, we have created three different versions of this workload, simulating low, medium and high load.

Chapter 5

Experimental Results

5.1 Workload Phase Analysis

The first thing we did before we evaluated the different schedulers was to examine the dynamic behavior of the workloads. We did that by executing a short-running version of each workload isolated on the server while monitoring its consumption of resources. The monitoring employed a very small time window, i.e. metrics taken once every second, in order to detect even short variations in resource usage. The results are depicted in figures [5.1] through [5.8] as percentage of the total resources available. When one or more metrics do not appear in the figure it means that their use by the workload was negligible. We observe that while the resource needs of the Hadoop benchmark [5.8] seem to vary greatly during its execution, a ten second average of these metrics is pretty stable. Moreover, the Media Streaming benchmark's variations, [5.5]-[5.7], have a fairly stable NetIO throughput and CPU utilization if we exclude the ramp-up and ramp-down phases in the beginning and the end of the workloads' execution. The latency critical, [5.3] and [5.4], and the other batch workloads, [5.1] and [5.2], showcase regular resource consumption. Hence, for the rest of our analysis we consider the workloads to have stable performance and resource usage.

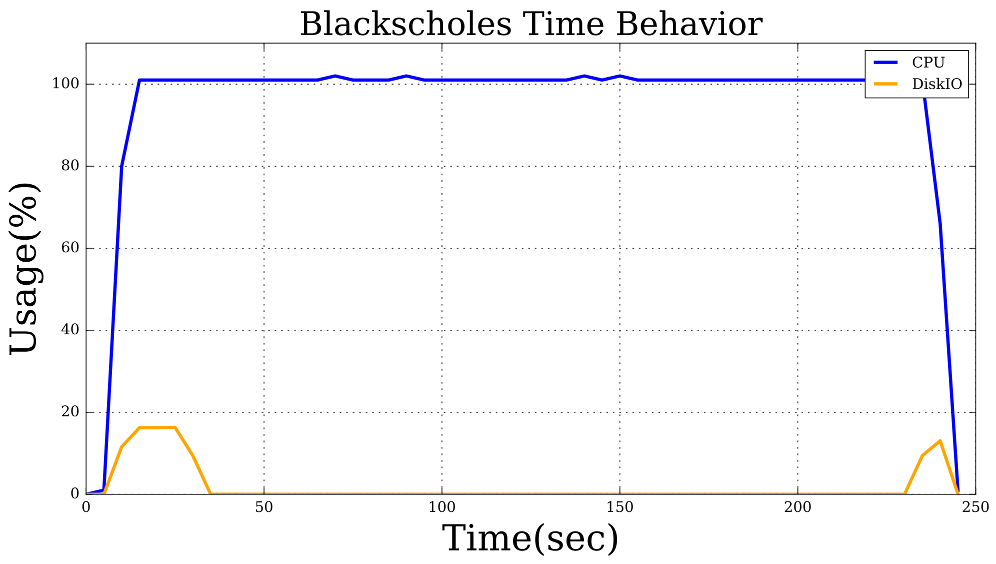


Figure 5.1: Blackscholes Time Behavior

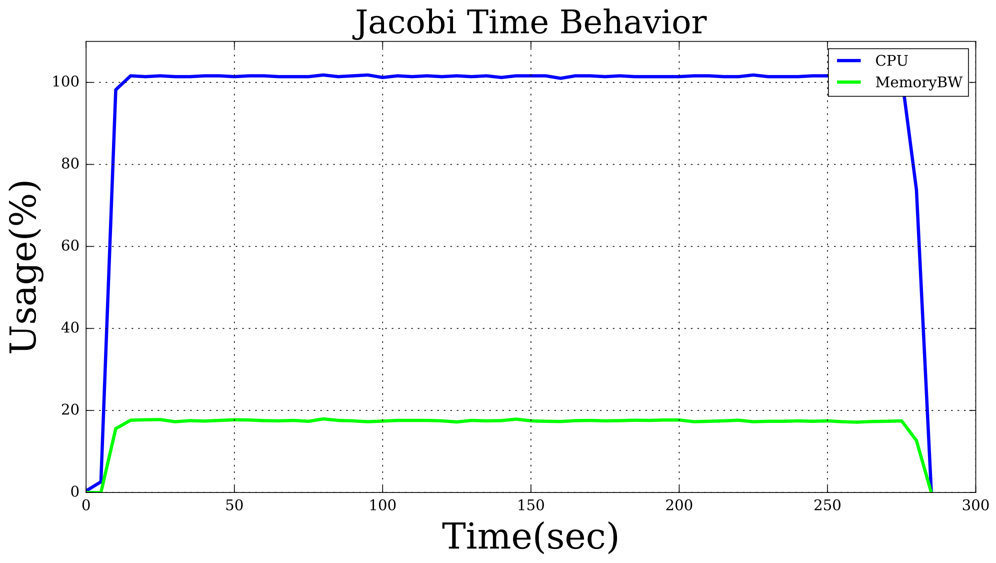


Figure 5.2: Jacobi Time Behavior

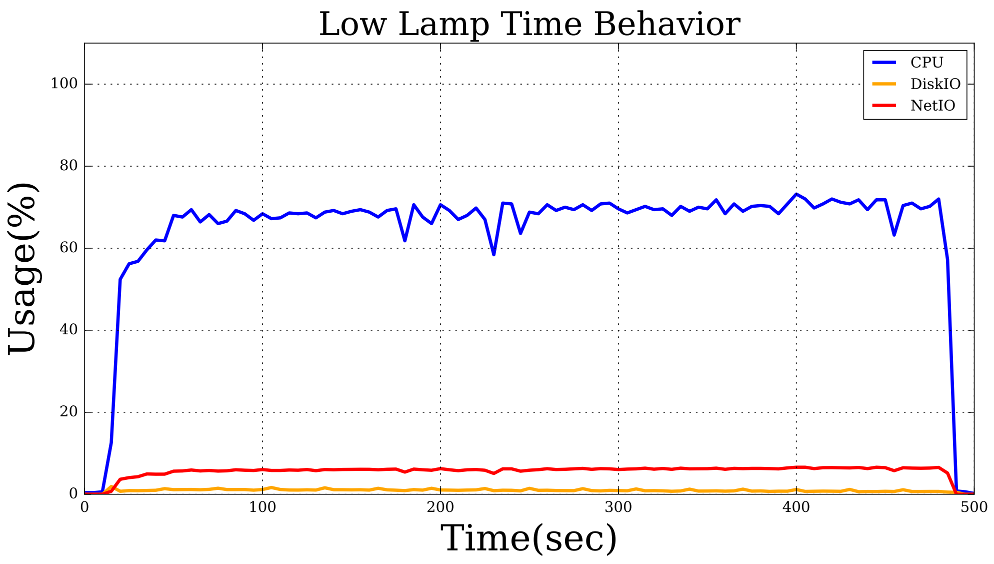


Figure 5.3: Low Load Lamp Time Behavior

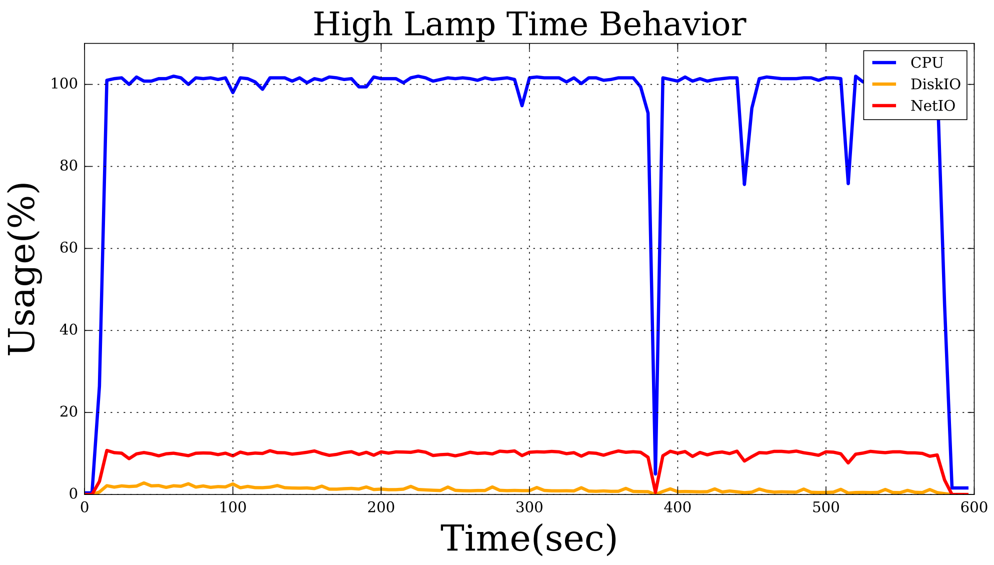


Figure 5.4: High Load Lamp Behavior

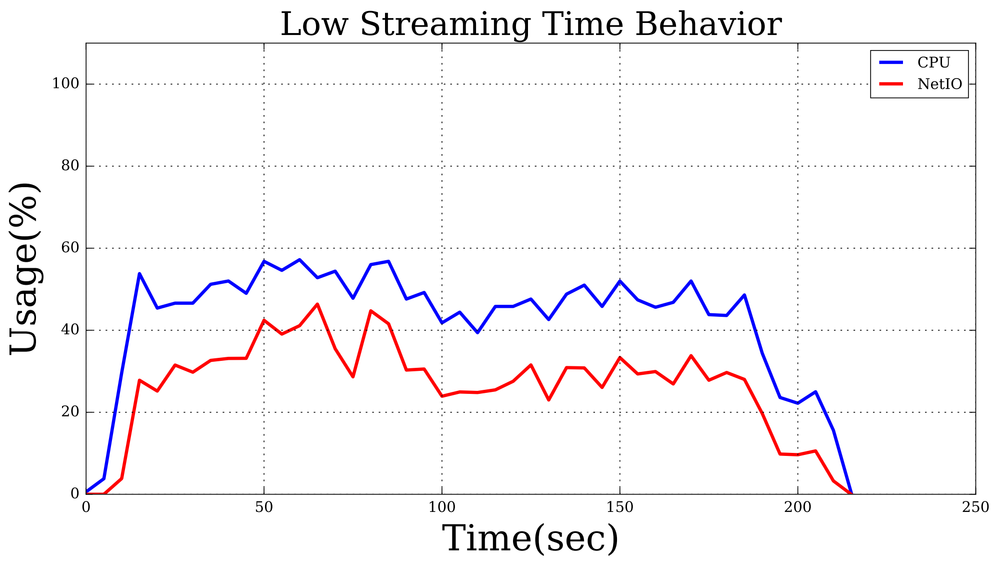


Figure 5.5: Low Load Media Streaming Time Behavior

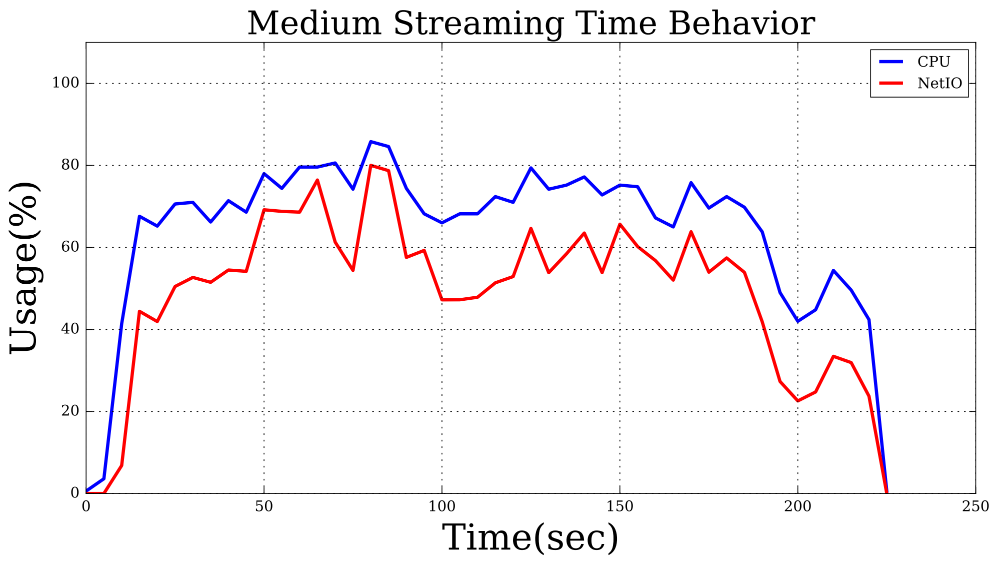


Figure 5.6: Medium Load Media Streaming Time Behavior

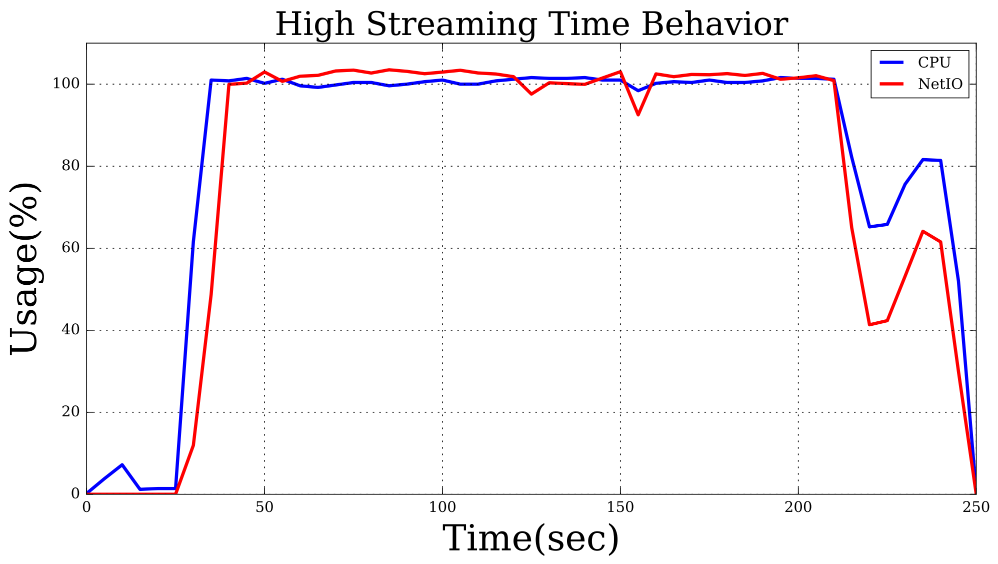


Figure 5.7: High Load Media Streaming Time Behavior

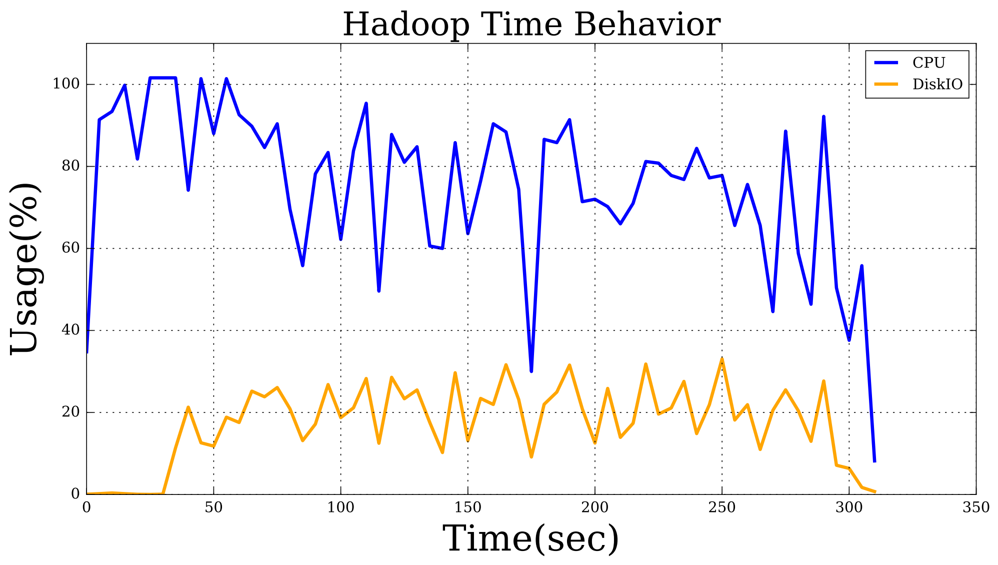


Figure 5.8: Hadoop Time Behavior

5.2 Model Building

The next step in our analysis was to do the necessary experiments in order to produce the data used by the schedulers. RBS needs the CPU, DiskIO, NetIO and Memory Bandwidth utilization of every workload as percentage of the total resources of the server when running alone on it. Using the monitoring system we obtained the average of these four metrics over the total running time of each workload and constructed the $N \times 4$ matrix $\mathbf{U}$[5.1], for our case $N = 8$. The results are as expected with the batch workloads having large CPU utilization while the media streaming and low latency workloads have NetIO and CPU utilization according to their load, with the latter showing a significantly lower need for network resources.

Resource Usage (%)				
Resources	Blackscholes	Jacobi	Low LAMP	High LAMP
CPU	100	100	59	96
Memory Bandwidth	0	18	0	0
NetIO	0	0	5.5	12
DiskIO	0	0	1	2
Resources	Low Streaming	Medium Streaming	High Streaming	Hadoop
CPU	36	66	73	76
Memory Bandwidth	0	0	0	0
NetIO	28	50	75	0
DiskIO	0	0	0	19

Table 5.1: **U** matrix used by RBS

IBS, on the other hand, needs much more extensive and difficultly obtainable information. It requires knowledge of the slowdown each job suffers when pinned on the same CPU with every other job, i.e. if we have N workloads (in our case $N = 8$) the $N \times N$ matrix $\mathbf{S}$ [5.2]. In order to obtain this information we first execute each workload alone on the server and measure its performance which we use as baseline. Then we executed the workloads in pairs, pinned on the same core and measure their performance again in comparison to the baseline. For example, if a batch workload's running time in isolation is sixty minutes and when co-scheduled with a Media Streaming job it becomes ninety minutes we say that it suffers a slowdown of 1.5. During this procedure we observed many interesting results. For instance, the low load streaming workload remains almost completely unaffected regardless of what type of VM is pinned next to it. On the other hand, high load streaming benchmark is very sensitive to interference as its performance easily degrades to a large degree. The Blackscholes, Jacobi and Hadoop benchmarks perform as expected from their resource usage while the low load LAMP is impervious to interference, at least when co-scheduled with just one other workload. However, the high load LAMP workload is very sensitive as its performance degrades severely regardless of the type of the co-located workload except for the low load LAMP.

In addition to the above, we executed the workloads again in pairs, but this time pinned on different cores in the same socket, and measured their performance in comparison to the baseline. As expected, their performance was affected when both were utilizing the same resource (e.g. two Media Streaming Workshops utilizing NetIO or two Hadoop workloads utilizing DiskIO) or their software stack (LAMP) was somehow sensitive to the utilization of a common resource such as LLC by the workload placed next to them. This information will be used in future work in order to further improve IBS.

Slowdown Time-Sharing								
Bench/Bench	Blackscholes	Jacobi	L. LAMP	H. LAMP	L. Streaming	M. Streaming	H. Streaming	Hadoop
CPU	2.00	2.02	1.84	1.92	1.42	1.71	1.66	1.96
Jacobi	2.05	2.08	1.98	2.00	1.54	1.70	1.66	2.04
Low LAMP	1.20	1.30	1.25	1.00	1.25	1.40	2.50	1.50
High LAMP	2.20	2.38	1.16	2.00	1.50	1.87	1.81	2.65
Low Streaming	1.00	1.00	1.03	1.02	1.05	1.08	1.44	1.02
Medium Streaming	1.10	1.11	1.00	1.05	1.07	1.18	1.72	1.20
High Streaming	2.08	2.19	1.2	2.11	1.19	1.31	1.72	1.93
Hadoop	1.69	1.74	1.42	1.55	1.29	1.45	1.41	1.74

Table 5.2: Time-sharing Slowdown: **S** matrix used by IBS

	Slowdown Space-Sharing							
Bench/Bench	Blackscholes	Jacobi	L. LAMP	H. LAMP	L. Streaming	M. Streaming	H. Streaming	Hadoop
CPU	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Jacobi	1.01	1.35	1.12	1.23	1.00	1.07	1.08	1.08
Low LAMP	1.00	1.00	1.00	1.00	1.00	1.00	1.47	1.2
High LAMP	1.04	1.31	1.03	1.05	1.02	1.05	1.10	1.68
Low Streaming	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Medium Streaming	1.00	1.00	1.00	1.00	1.00	1.06	1.11	1.00
High Streaming	1.00	1.00	1.05	1.07	1.30	1.57	1.56	1.05
Hadoop	1.00	1.08	1.02	1.01	1.00	1.00	1.05	1.35

Table 5.3: Space-Sharing Slowdown

5.3 Schedulers' Performance

The schedulers were evaluated through the execution of three diverse scenarios. They consist of batch and latency-critical workloads which are large consumers of resources on private and public clouds.

5.3.1 Random Scenario

The first scenario we examine is a random scenario containing all types of workloads. The server is shared between batch, media streaming and latency critical benchmarks. This is a must-examined scenario in order to evaluate the schedulers' performance in unknown and unexpected conditions. The workloads arrive in the server with 30-sec inter-arrival time. We define as Subscription Ratio (SR) the ratio of the number of jobs placed in the server to the number of cores the server has, i.e. in our case (12-core server) a subscription ratio of 0.5 means that 6 jobs are placed in the server while a ratio of 2 shows that we have placed 12 jobs. The metrics we evaluate are the average performance of all workloads of the scenario compared to their performance in an isolated environment (represented by 100 in the scale) and the core minutes that the total scenario run consumes. As the scenario is random all algorithms are expected to perform mediocre in terms of performance. When the server is underscribed, i.e. $SR \leq 1$, RBS and IBS should result to large savings in core minutes suffering only minimal performance degradation over RRS. The experimental results support this estimation as for $SR = 0.5$ RBS saves 32% in cores minutes with only 2.5% performance degradation over RRS while IBS saves too 32% in core minutes suffering though 7% performance degradation. For $SR = 1$ IBS performs better, reducing by 34.5% core minutes while allowing the performance of the workloads degrade by 8.5%, with RBS offering an improvement of 24% with 8% performance degradation. When the server is oversubscribed we expect RBS and IBS to produce better performance due to better selection of the workloads that will time-share as time-sharing is necessary for all schedulers in this case and not just introduced in order to save core minutes. Further gains for IBS and RBS are expected from their ability to detect idle workloads and rearrange the running ones in order to improve their performance and free cores. These gains should be important in this scenario, especially regarding the core minutes consumed, due to the diversity of the workloads used and the subsequent variation in their finish times that allows the detection of idle workloads to yield significant improvements. For $SR = 1.5$ we observe improvements for RBS and IBS over RRS both in core minutes ($30\% - 35\%$) and performance (10%). However, for $SR = 2$ the performance is about stable with the core minutes improving 11.5% (RBS) to 15% (IBS). This behavior can be explained by the fact that in this case the server is severely oversubscribed with the workload placement not mattering so much in terms of performance while the small improvements to core minutes consumption can be attributed to the detection and consolidation of idle workloads.

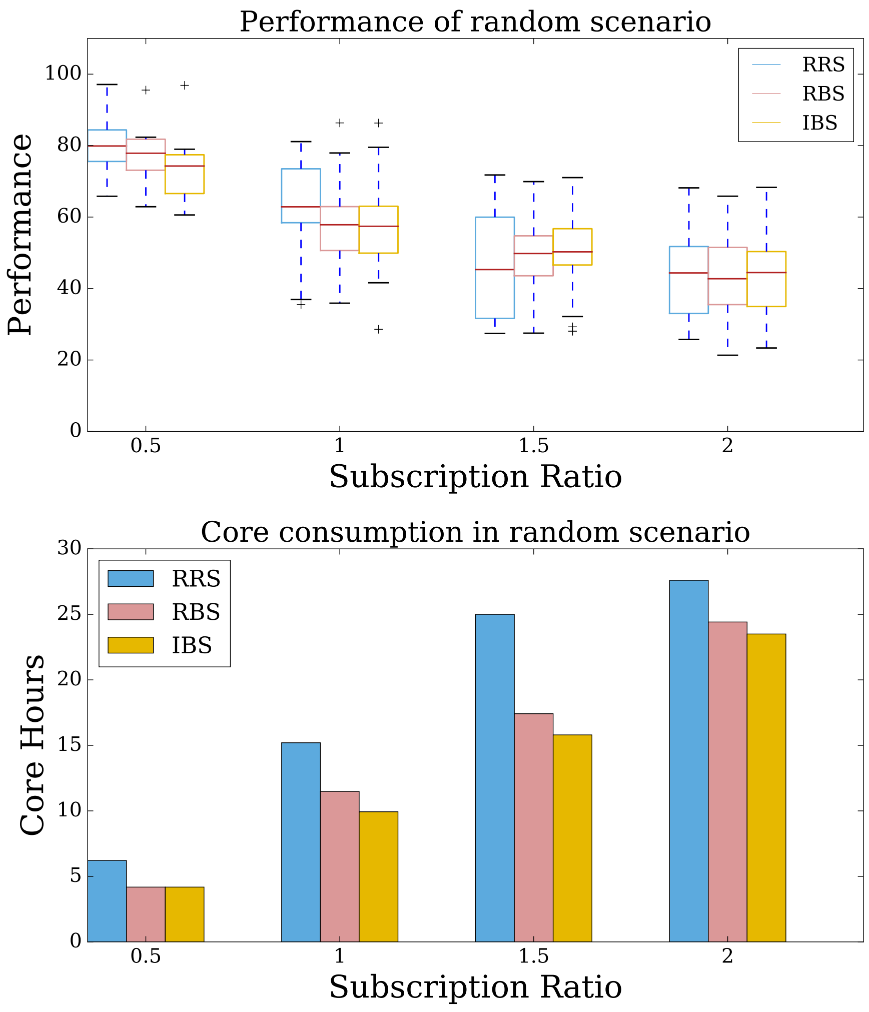
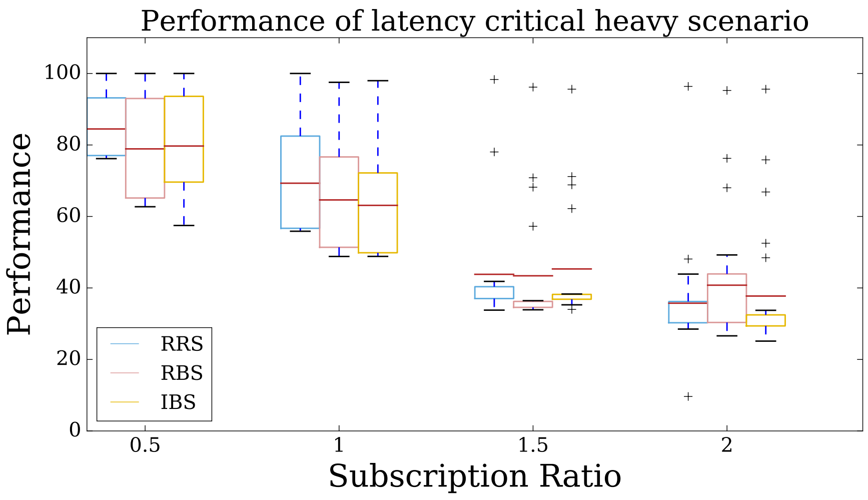


Figure 5.9: Random Scenario

5.3.2 Latency-critical Heavy Scenario

Latency-critical services are major tenants in cloud facilities. As a result of this, a skewed workload distribution has been observed in cloud services as most users host latency-critical but low load applications, as modeled in this scenario. The scenario also includes a small number of batch workloads and a single low load media streaming workload. The latency critical services are more sensitive both to time- and space-sharing interference in comparison to batch and media streaming benchmarks. Due to the low load it is possible for RBS and IBS to consolidate jobs in less cores than RRS for SR up to 1.5. This leads to a significant reduction of core minutes consumption of at least 30% and up to 50% for IBS in $SR = 1$ with the performance degradation never exceeding 10%. For $SR = 2$ RBS attains a 10.5% reduction in core minutes consumed, together with a 14% improvement in performance by avoiding the co-scheduling of workloads that behave really bad together. IBS, due to its aggresive nature, provides an increased 25% core minute improvement, however accompanied by a lower 5.5% performance improvement over RRS.



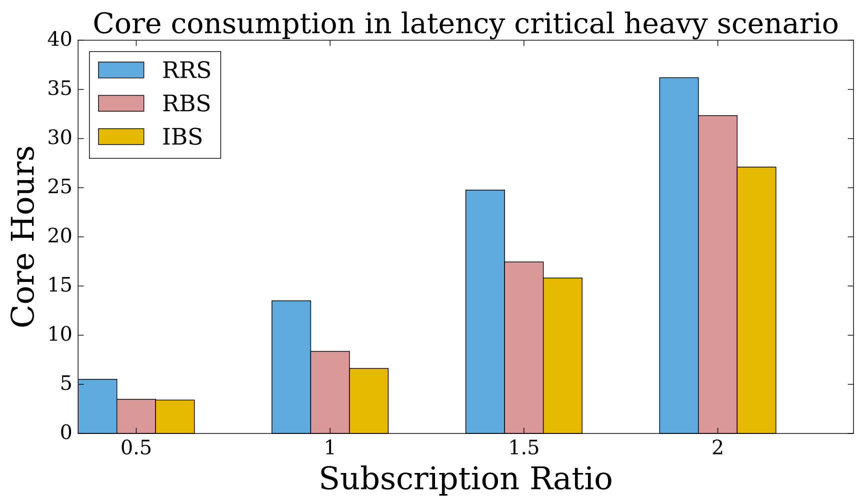


Figure 5.10: LC Scenario

5.3.3 Dynamic Scenario

Usage of cloud services is time-dependent as VMs go through execution phases. For example, a VM used both for development and deployment of an application is expected to have very low resource requirements during development and high consumption during deployment. Furthermore, the load of internet services varies depending on the time of day, e.g. low traffic in the morning vs high traffing in the evening, and/or the date, e.g. low traffic during holiday periods. In order to model this behavior we designed a scenario where 24 random VMs are placed in the server where they become active in 12- or 6-job batches. RRS, being unaware of the monitoring system's metrics and making static decisions about the pinning, needs to reserve the whole server continuously regardless of the state of the VMS (idle/running). On the other hand, RBS achieves around 18% improvement in peformance by avoiding the time-sharing of active and sensitive workloads while leaving a large number of cores unused due to the detection and consolidation of idle workloads. IBS consolidates workloads even more aggresively using less cores than RBS but with the offset of a reduced (13%) improvement of their performance.

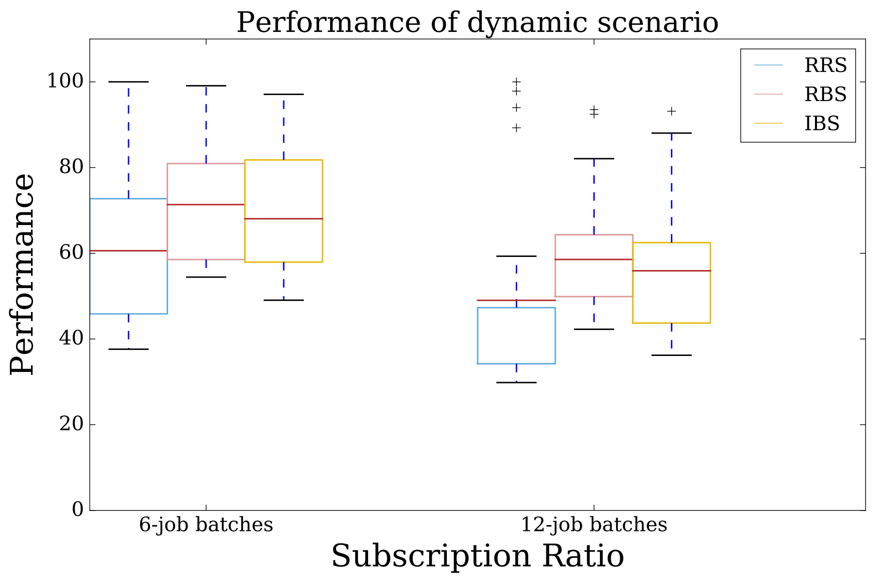
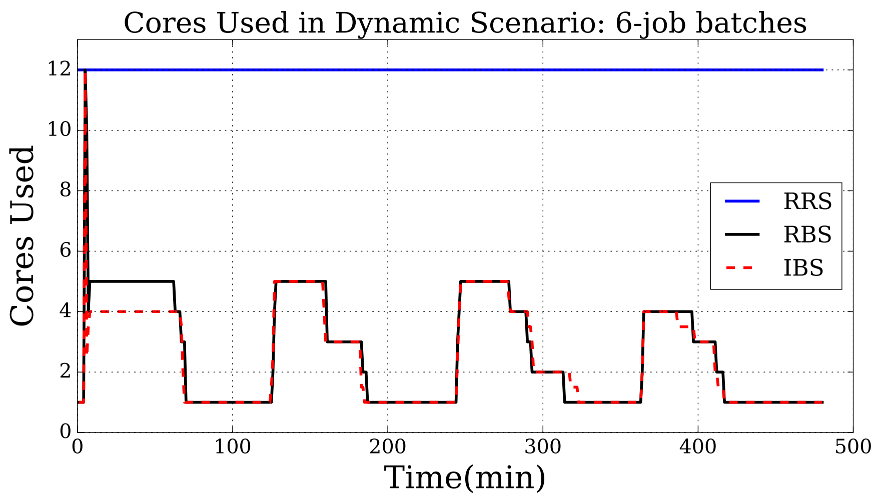


Figure 5.11: Boxplot of the DS performance.



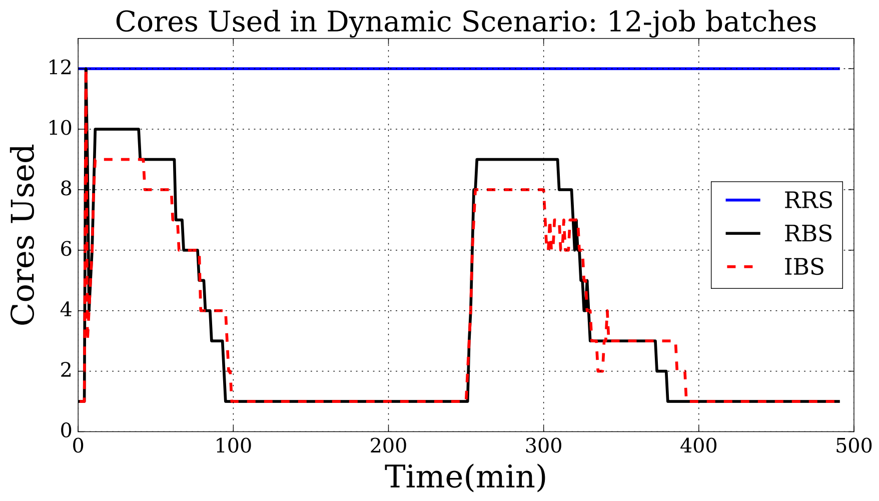


Figure 5.12: Core usage in DS.

Chapter 6

Discussion

6.1 Results' Evaluation

In order to evaluate the performance of the scheduling algorithms presented in the previous chapter, we consider the amount of on-host CPU time saved through VM consolidation over the workload performance variation (slowdown or speedup). Host CPU time can be construed as a measure of the total resource usage on the host, and thus lower CPU consumption relates to more efficient use of resources. Also, since most of the fraction of electricity consumed by a running server is proportionate to CPU utilization, lowering CPU utilization is also beneficial in terms of energy consumption. Thus, evaluating the gains in terms of efficiency vs the performance trade-off will accurately highlight better performing schedulers in light of the main performance indicator for VM consolidation.

To measure scheduler performance we use:

$$M = \frac{(P_{ALG} - P_{RRS})}{P_{RRS}} + \frac{(C_{RRS} - C_{ALG})}{C_{RRS}}$$

where M is the scheduler performance metric, P is the average performance of all workloads scheduled (using RRS or some other scheduling ALGorithm) and C is the CPU time consumed by all the co-scheduled workloads when run to completion. The metric is designed to take into account the gains in performance and efficiency in conjunction and penalize severe performance degradation with regards to the simple round-robin scheduler.

With regards to the experimental results presented in the previous chapter, we can conclude that both the Threshold-Based Scheduler (RBS) and Interference-Based

Table 6.1: Metric evaluation of algorithms vs RRS

SR	RBS RS	IBS RS	RBS LS	IBS LS
0.5	0.3017322721	0.2569134853	0.3016877948	0.3262783336
1	0.1645904566	0.2598010751	0.3119356019	0.4184069583
1.5	0.4024503311	0.4777130243	0.285943619	0.3956368481
2	0.07868456622	0.1507817439	0.2476092701	0.3072840804

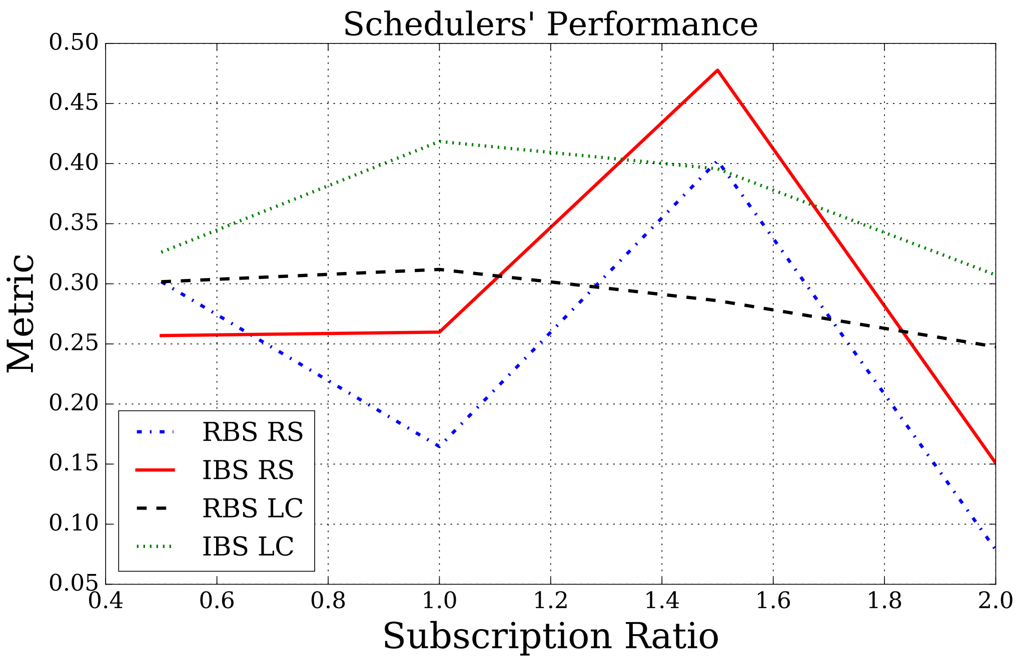


Figure 6.1: Metric evaluation of RBS and IBS scheduling performance.

Scheduler (IBS) outperform round-robin placement of VMs on the compute host in terms of resource consumption and, in the over-subscribed cases, in performance as well. Considering our composite metric, with results depicted in Table 6.1 and Figure 6.1 for graphic comparison, the performance of both schedulers peaks for SR around 1-1.5, i.e. when the server is conservatively oversubscribed. Interference-based scheduling seems to significantly outperform RBS in all cases, except for $SR = 0.5$ in the random scenario. The schedulers' performance is minimum for $SR = 2$ as in that case there are few things they can do in order to improve performance and reduce core minutes consumption other than consolidate idle workloads. Whatever the pinning is, for $SR = 2$, the performance degradation that workloads suffer is severe.

Although the 120% setting used in our experiments limits RBS, it is selected in order to provide the best results of this scheduler class given our experimental setup. IFS seems more aggressive in our results but this is an artifact of the workload selection and amount of interference allowed. IFS can vary the intensity of VM consolidation depending on the way chosen to avoid workload interference.

6.2 Conclusions

In this thesis we implement and evaluate a number of scheduling algorithms that range from round-robin, to resource-based, to workload interference-aware scheduling. We test these algorithms using three realistic scenarios on a host and measure their performance in terms of preserving VM QoS and decreasing overall host utilization. Our approach treats the global, DC-level VM consolidation problem as a set of discreet optimizations for the placement of VM workloads on each physical host.

We examine the effects of VM oversubscription on workload QoS and show that by taking into account workload interference both host efficiency and VM performance can be improved. Also, our experimental results show that VM consolidation can, if performed with care, be acceptable in terms of performance degradation even for latency-critical applications.

To extend our work, further examination of resource-based and interference-aware schedulers for larger subscription ratios is planned, in order to validate the savings observed in our experiments to a wider range of scenarios. We also plan to make a wider exploration of local vs global consolidation approaches using a private cloud to pit our approach against infrastructure-scale schedulers and take into account the space-sharing interference between workloads.

Bibliography

[1] Hadoop Architecure. http://blogs.sitepointstatic.com/.

[2] Amazon EC2. https://aws.amazon.com/ec2/.

[3] Windows Azure. https://azure.microsoft.com/en-us/.

[4] Google Compute Engine. https://cloud.google.com/compute/.

[5] Openstack. https://www.openstack.org/.

[6] VMware vCloud. https://www.vmware.com/products/vcloud-suite.

[7] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[8] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan and Claypool Publishers, 1st edition, 2009.

[9] Ripal Nathuji, Canturk Isci, and Eugene Gorbatov. Exploiting platform heterogeneity for power efficient data centers. In *Fourth International Conference on Autonomic Computing, Jacksonville, Florida, USA, June 11-15, 2007*, page 5, 2007.

[10] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. Technical Report MSR-TR-2011-55, May 2011.

[11] Christina Delimitrou and Christos Kozyrakis. QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon. In *ACM Transactions on Computer Systems (TOCS)*, 2013.

[12] Christina Delimitrou and Christos Kozyrakis. iBench: Quantifying Interference for Datacenter Workloads. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC)*, September 2013.

[13] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, New York, NY, USA, 2014. ACM.

[14] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 450–462, 2015.

[15] Alan Roytman, Aman Kansal, Sriram Govindan, Jie Liu, and Suman Nath. Pacman: Performance aware virtual machine consolidation. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 83–94, San Jose, CA, 2013. USENIX.

[16] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 219–230, Berkeley, CA, USA, 2013. USENIX Association.

[17] Sogand Shirinbab and Lars Lundberg. Performance implications of over-allocation of virtual cpus. In *Networks, Computers and Communications (ISNCC), 2015 International Symposium on*, pages 1–6. IEEE, 2015.

[18] Davide Adami, Barbara Martini, Andrea Sgambelluri, Molka Gharbaoui, Piero Castoldi, Christian Callegari, Lisa Donatini, and Stefano Giordano. Cloud and network service orchestration in software defined data centers. In *Performance Evaluation of Computer and Telecommunication Systems (SPECTS), 2015 International Symposium on*, pages 1–6. IEEE, 2015.

[19] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014.

[20] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX*

Conference on Networked Systems Design and Implementation, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[21] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13, New York, NY, USA, 2012. ACM.

[22] Christina Delimitrou and Christos Kozyrakis. The Netflix Challenge: Datacenter Edition. Los Alamitos, CA, USA, July 2012. IEEE Computer Society.

[23] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SOCC)*, August 2015.

[24] Fei Guo, Hari Kannan, Li Zhao, Ramesh Illikkal, Ravi Iyer, Don Newell, Yan Solihin, and Christos Kozyrakis. From chaos to qos: Case studies in cmp resource management. *SIGARCH Comput. Archit. News*, 35(1):21–30, March 2007.

[25] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, Santa Clara, CA, July 2015. USENIX Association.

[26] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM.

[27] Andrej Podzimek, Lubomír Bulej, Lydia Y. Chen, Walter Binder, and Petr Tuma. Analyzing the impact of CPU pinning and partial CPU loads on performance and energy efficiency. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*, pages 1–10, 2015.

[28] Ganglia. http://ganglia.sourceforge.net/.

[29] libvirt. http://libvirt.org/.

[30] perf. https://perf.wiki.kernel.org/.

[31] Hui Wang, Canturk Isci, Lavanya Subramanian, Jongmoo Choi, Depei Qian, and Onur Mutlu. A-DRM: architecture-aware distributed resource management of

virtualized clusters. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Istanbul, Turkey, March 14-15, 2015*, pages 93–106, 2015.

[32] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[33] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[34] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 37–48, New York, NY, USA, 2012. ACM.