Master Thesis

June 6, 2017

Cloud Benchmarking

Estimating Cloud Application Performance Based on Micro Benchmark Profiling

Joel Scheuner

of Muensterlingen, Switzerland (10-741-494)

supervised by

Prof. Dr. Harald C. Gall

Dr. Philipp Leitner

University of Zurich[UZH]

s.e.a.l. software evolution & architecture lab

Master Thesis

Cloud Benchmarking

Estimating Cloud Application Performance Based on Micro Benchmark Profiling

Joel Scheuner

University of Zurich[UZH]

s.e.a.l. software evolution & architecture lab

Master Thesis

Author: Joel Scheuner, joel.scheuner@uzh.ch

URL: `https://github.com/joe4dev`

Project period: 06.12.2016 - 06.06.2017

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to offer special thanks to Philipp Leitner for his guidance, valuable feedback, and overall support during my master thesis as well as over the course of the past three years of joint research. I'm very much looking forward to continuing this successful collaboration with my prospective Ph.D. advisor in Gothenburg at Chalmers University. My sincere thanks go to Prof. Harald Gall for supporting me as a member of the s.e.a.l. research group at University of Zurich and providing a enriching environment for work and research.

I express my deepest gratitude to all the people who accompanied and supported me on the way, filled with inspiring learning and joyful collaboration, towards this thesis and the completion of my master studies. My personal thanks go to my parents for supporting me in uncountable ways throughout my life. My dear friends, best teammates, and fellow students enriched the past years of study and life in a manner to keep in best memories. Thank you very much Domi, Genc, Seebi, Fabio, Moritz, Beni, Martina, Matthias, Greggy, Ale! Your ongoing motivation, inspiring discussions, and refreshing breaks were invaluable. I say thank you to Seebi, Chrigi, and Yao for their proofreading efforts. Moreover, thanks to all s.e.a.l.s for your inputs and collegially integrating me into the group. Last but not least, thank you very much my dearest girlfriend Yao for motivating and supporting me.

Abstract

The continuing growth of the cloud computing market has led to an unprecedented diversity of cloud services. To support service selection, micro benchmarks are commonly used to identify the best performing cloud service. However, it remains unclear how relevant these synthetic micro benchmarks are for gaining insights into the performance of real-world applications.

Therefore, this thesis develops a cloud benchmarking methodology that uses micro benchmarks to profile application performance and subsequently estimates how an application performs on a wide range of cloud services. A study with a real cloud provider has been conducted to quantitatively evaluate the estimation model with 38 selected metrics from 23 micro benchmarks and 2 applications from different domains. The results reveal remarkably low variability in cloud service performance and show that selected micro benchmarks can estimate the duration of a scientific computing application with a relative error of less than 10% and the response time of a Web serving application with a relative error between 10% and 20%. In conclusion, this thesis emphasizes the importance of cloud benchmarking by substantiating the suitability of micro benchmarks for estimating application performance but also highlights that only selected micro benchmarks are relevant to estimate the performance of a particular application.

Zusammenfassung

Das anhaltende Wachstum des Cloud Computing Marktes führte zu einer noch nie dagewesenen Vielfalt an Cloud-Diensten. Bei der Entscheidungsunterstützung zur Wahl des leistungsstärksten Cloud-Dienstes werden verbreitet Micro Benchmarks eingesetzt, wobei unklar bleibt wie relevant diese künstlichen Micro Benchmarks sind, um Erkenntnisse über die Leistung von realen Anwendungen zu gewinnen.

Daher wird in dieser Arbeit eine Methode zum Leistungsvergleich von Cloud-Diensten entwickelt, welche Micro Benchmarks verwendet um ein Leistungsprofil einer Anwendung zu erstellen und damit die Leistung dieser Anwendung über eine grosse Auswahl an Cloud-Diensten abzuschätzen. Eine Studie mit einem realen Cloud-Anbieter wurde durchgeführt, um das Schätzungsmodell mit 38 ausgewählten Metriken von 23 Micro Benchmarks und 2 Anwendungen aus unterschiedlichen Domänen quantitativ zu evaluieren. Die Resultate zeigen, dass sich die Leistung von Cloud-Diensten bemerkenswert stabil verhält und dass ausgewählte Micro Benchmarks die Laufzeit einer wissenschaftlichen Computing-Anwendung mit einem relativen Fehler von weniger als 10% und die Antwortzeit einer Webserver-Anwendung mit einem relativen Fehler zwischen 10% und 20% abschätzen können. Zusammenfassend betont diese Arbeit die Wichtigkeit von Leistungsvergleichstests in der Cloud durch das Nachweisen der Eignung von Micro Benchmarks zum Abschätzen der Leistung einer Anwendung. Allerdings zeigt diese Arbeit auch auf, dass nur ausgewählte Micro Benchmarks für die Leistungsschätzung einer bestimmten Anwendung relevant sind.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Cloud computing [AFG+09, BYV+09, MG11] fundamentally changes the way how computing services are provisioned. It offers computing resources (*e.g.*, Virtual Machines (VMs)[1] on Amazon's Elastic Compute Cloud), programming environments (*e.g.*, Ruby on Heroku[2]), or entire applications (*e.g.*, business apps on Google Suite[3]) as on-demand utilities on a pay-per-use basis. The revolutionary effect of the disruptive cloud computing paradigm is repeatedly mentioned in recent literature [You17, BdDPP16, EGHO16] and reputable technology analyst reports [PG17, BNC+16]. Beyond becoming "one of the hottest topics in the field of information systems" [WLJ+16] for academics, cloud computing surpasses predicted growth rates [BNC+16] reaching a total market of over $200 billion in 2016 [PG17]. Some analysts even forecast that cloud computing could reach the same ubiquity as internet connectivity at the end of this decade [FvdM16] with large companies shifting their strategies from *cloud-first* [Nad14] to *cloud-only* [FvdM16].

In today's fastest growing service model called Infrastructure-as-a-Service (IaaS) [PG17, BNC+16], computing resources, such as CPU processing time, disk space, or networking capabilities, can be acquired and released as self-service via an Application Programming Interface (API) prevalently in the form of VMs. Such VMs are typically available in different configurations or sizes also known as instance types, machine types, or flavors. This diversity ranges from tiny-sized VMs with less than 1 (shared) CPU core and 1GB RAM (*e.g.*, *f1-micro*[4]) to super-sized VMs with 128 CPU cores and 1952 GiB RAM (*e.g.*, *x1.32xlarge*[5]). The $25 billion IaaS market (as of 2016) [PG17] is further extending its offers headed by the three leading IaaS providers [Dor16] Amazon Web Services (AWS) Elastic Compute Cloud (EC2)[1], Microsoft Azure Virtual Machines[6], and Google Cloud Platform Compute Engine[7].

Given the large service diversity, selecting an appropriate VM configuration for an application is a non-trivial challenge. While functional properties can be compared by studying provider information or using tools such as Cloudorado[8], non-functional properties, such as performance, need to be measured tediously. The field of research called cloud benchmarking is devoted to objectively measure and compare the differences in performance between the various cloud services. A large body of literature [LC16, IOY+11, OIY+10, SDQR10, FJV+12, OZL+13, Wal08] reports performance measurements for different workloads at the very resource-specific (*e.g.*, CPU integer operations) and artificial micro-level or at the domain-specific (*e.g.*, Web serving) and real-world application-level.

[1] `https://aws.amazon.com/ec2/`
[2] `https://devcenter.heroku.com/categories/ruby`
[3] `https://gsuite.google.com/`
[4] `https://cloud.google.com/compute/docs/machine-types`
[5] `https://aws.amazon.com/ec2/instance-types/x1/`
[6] `https://azure.microsoft.com/en-us/services/virtual-machines/`
[7] `https://cloud.google.com/compute/`
[8] `https://www.cloudorado.com/cloud_providers_comparison.jsp`

Existing literature largely focuses on either application benchmarks or micro benchmarks in isolation. Researchers propose new cloud-specific application benchmarks [FAK+12, PSF16] and evaluate their performance [IYE11, DPC10, BLL+14] in cloud environments. However, application benchmarks tend to require an elaborate setup, run over a long time, and deliver polysemous results with multiple metrics. The high benchmarking effort lets researchers resort to micro benchmarks, which are typically easy to install, quick to run, and clear to interpret as single metrics. Extensive studies have been conducted to collect performance measurements for many different VM configurations [LC16, VAM+16, IOY+11, OIY+10]. However, it remains unclear how relevant these artificial benchmarks are to gain insights about the performance of real-world applications.

1.1 Goals and Contributions

The goal of this thesis is to investigate the suitability of micro benchmarks for estimating cloud application performance across different instance types. Consequently, the following Research Questions (RQs) are addressed:

RQ1 – Performance Variability *within Instance Types*

Does the performance of equally configured cloud instances vary relevantly?

This *intra-instance type* performance variability is relevant for the estimation of application performance. While high variability could favor (if correlated) or hamper (if random) meaningful estimates, low variability could facilitate inter-instance type estimation. A correlated high variability (*i.e.*, different workloads are all either slow or fast) indicates the existence of slower and faster instances. Estimating this fitness value (*i.e.*, good or bad instance) could be exploited by placement gaming strategies as proposed in [OZN+12, OZL+13, FJV+12]. Conversely, a random high variability makes it very hard to find any significant patterns. Finally, low variability reduces the sample size required to make more accurate and confident estimates.

Knowing the nature of performance variability motivates the subsequent research question:

RQ2 – Application Performance Estimation *across Instance Types*

Can a set of micro benchmarks estimate application performance for cloud instances of different configurations?

RQ2 aims towards verifying whether it is possible to build a meaningful model that estimates application performance for cloud instances of *different configurations* (*i.e.*, different instance types) using micro benchmark profiling. This research question is divided into the following two sub-questions:

RQ2.1 – Estimation Accuracy

How accurate can a set of micro benchmarks estimate application performance?

This sub-question addresses the estimation accuracy of application performance for previously unseen cloud instance configurations based on the trained model using a set of micro benchmarks. The performance of such previously unseen cloud instance types could deviate by factors higher than 2 and therefore rough applications performance estimates exhibiting relative errors of 10-20% can still be beneficial compared to having no guidance during instance type selection. This is particularly true when facing a large choice of potential instance types.

To optimize the estimation model for practical applicability, the profiling effort should be restricted to a minimal set of relevant micro benchmarks. This feature selection process is tackled by the subsequent sub-question:

RQ2.2 – Micro Benchmark Selection

Which subset of micro benchmarks estimates application performance most accurately?

This sub-question also investigates whether some micro benchmarks can be used interchangeably with marginal loss of estimation accuracy. Such substitution flexibility could further reduce the profiling effort by minimizing the execution time and the number of micro benchmarking tools to install.

In order to answer these questions, a cloud benchmarking study has been designed, implemented, and conducted in a real cloud computing environment. Hereby, a Web serving application benchmark has been crafted from ground up and fully automated using Cloud WorkBench [SLCG14, SCLG15] together with existing micro benchmarks.

This thesis makes the following five contributions:

1. It extends Cloud WorkBench (CWB) [SLCG14, SCLG15] with a modular benchmark plugin system.
2. It presents a newly crafted Web serving application benchmark with three different load scenarios.
3. It provides an automated CWB benchmark that combines single-instance and multi-instance micro and application benchmarks.
4. It reports a raw and cleaned data set with cloud benchmarking results from Amazon EC2.
5. It evaluates an estimation model for application performance based on micro benchmark profiling.

1.2 Thesis Outline

The remainder of this thesis is structured as follows: Chapter 2 introduces cloud computing, necessary fundamentals of systems performance benchmarking for the understanding of this thesis, and tools to automate cloud benchmarking. Chapter 3 presents related work in the areas of cloud performance variability, micro benchmarking, application benchmarking, and cloud instance type selection. Chapter 4 presents the methodology developed in this thesis to combine micro and application benchmarks, expounds the configuration of micro benchmarks, details about the design of the Web serving benchmark, describes the data pre-processing pipeline, and discusses the threats to validity of the presented methodology. Chapter 5 introduces the benchmarking data set and presents, discusses, and summarizes the results guided by the previously introduced research questions. Finally, Chapter 6 summarizes the contributions, concludes this thesis, and outlines future work.

Chapter 2

Background

This chapter introduces the necessary background for the understanding of this thesis in the areas of cloud computing, systems performance benchmarking, and cloud benchmarking automation.

2.1 Cloud Computing

Cloud computing [VRMCL08, Hil09, AI10, BBG11] is most commonly defined as:

> a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.
>
> —The National Institute of Standards and Technology (NIST) Definition [MG11]

The most important literature in the field [AFG$^+$09, BYV$^+$09, MG11] recognizes that the concept of cloud computing is not completely new and emerged from similar distributed systems paradigms such as utility computing, grid computing [FE04, FZRL08], and cluster computing. The essential characteristics of cloud computing [MG11] have been strongly influenced by the business model of utility computing, where computing resources are offered as a service and charged based on the actual usage, also known as *pay-per-use* or *pay-as-you-go*. This model has been envisioned already in the 1960s [Mag09] and is commonly referred to as delivering computing resources in a similar manner than traditional metered utilities such as electricity or water. Technologically, cloud computing is inspired by the grid computing paradigm, which realizes the idea of connecting commodity hardware to provide computing power at large scale as an alternative to supercomputers since the mid 90s [FZRL08]. However, grid computing became never commoditized outside the high-performance computing community. Furthermore, the emergence of the cloud computing terminology in 2007[1] led to clearer differentiation [FZRL08, AFG$^+$09, BYV$^+$09, MG11] such that AWS is no more called grid computing [Gar07]. In contrast to more heterogeneous and widely distributed grids, cloud computing constitutes the emergence of super large scale data centers (>50000 servers) connected through fast Local Area Networks (LANs) [AFG$^+$09]. This resembles the principle of cluster computing, where a typically heterogeneous set of interconnected servers (*i.e.*, nodes) is treated as a single integrated computing resource [BYV$^+$09]. The novelty of cloud computing is the combination, refinement, and extension of ideas from existing paradigms to offer illusionary unlimited computing resources [AFG$^+$09] from the pool of massive multi-tenant data centers [AFG$^+$09, ZCB10] as fully automated [MG11, ZL11] and almost instantly available [AFG$^+$09] self-service [MG11] that is charged per usage and made available over the internet [ZCB10].

[1] `https://trends.google.com/trends/explore?date=all&q=cloud%20computing,grid%20computing`

2.1.1 Service Models

Figure 2.1 illustrates the 3 service models of cloud computing [MG11] offering resources at different levels of abstraction.

1. IaaS offers low-level compute (*e.g.*, EC2[2]), storage (*e.g.*, Elastic Block Storage (EBS)[3]), and network resources (*e.g.*, Virtual Private Cloud (VPC)[4]). These resources are most commonly provided in the form of VMs where users nearly fully control the entire software stack [KSHD13].

2. Platform-as-a-Service (PaaS) offers a provider-managed environment for building and deploying applications in the cloud (*e.g.*, Heroku[5]) [HK10] where users control their applications and data but have no immediate access to the underlying environment software and hardware infrastructure [MG11].

3. Software-as-a-Service (SaaS) offers fully functional applications (*e.g.*, Google Suite[6]) where users have no more profound control than what is exposed in the built-in application settings [MG11].

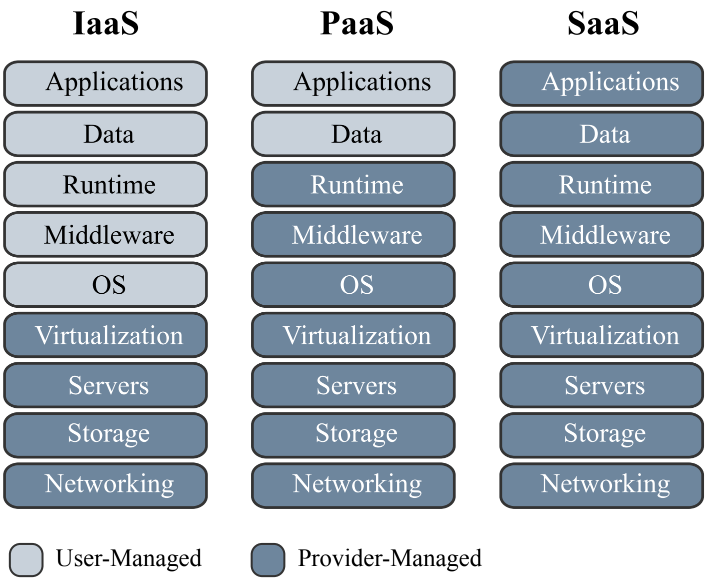

Figure 2.1: Abstractions in Cloud Computing Service Models (Adapted[7] from [Dav16])

[2]https://aws.amazon.com/ec2/
[3]https://aws.amazon.com/ebs/
[4]https://aws.amazon.com/vpc/
[5]https://www.heroku.com/
[6]https://gsuite.google.com/

2.1.2 Cloud Infrastructure

Cloud providers typically organize their global infrastructures into geographically distributed regions with discrete data centers. The geographical regions are vaguely described with names of geographical regions (*e.g.*, West US[8]), countries (*e.g.*, Ireland[9]), or states (*e.g.*, Iowa[10]) and sometimes identified by an API names such as *eu-west-1* (Ireland) or *us-central1* (Iowa). Regions are strategically chosen to be in close proximity to customers and the exact data center placement is often driven by environmental factors such as low energy cost [AFG+09] because power consumption is a dominating cost factor in cloud data centers [Ham09]. Large providers such as AWS or Google Cloud Platform operate discrete data centers within the same region to achieve high availability through redundant infrastructure with support for fail-over. This concept is typically known as Availability Zone (AZ) and denoted by alphabetical suffixes in region identifiers (*e.g.*, *eu-west-1a* or *us-central1-a*).

2.1.3 Virtual Machine Resources

Virtual machines are offered with varying computing capabilities regarding different resources. Cloud providers usually specify a set of preconfigured *instance types*[11], also known as machine types[12], flavors, or VM sizes[13]. These instance types differ in their resource characteristics such as the number of virtual Central Processing Units (CPUs) (vCPUs), the amount of Random-Access Memory (RAM), the level of network performance (*e.g.*, low, medium, high), and the type of Input/Output (I/O) (*e.g.*, Hard Disk Drive (HDD), Solid State Disk (SSD)). The concrete instantiation of a particular instance type is called *instance* and obtained whenever the cloud user acquires a new VM. Notice that instances typically do not have their own local instance storage but are connected to a dedicated storage service such as EBS.

Instance types are categorized into families of specialized resources for different use cases. General purpose instance types are designed for a wide range of used cases with a well-balanced resource profile. Optimized instance types are specifically equipped for compute-heavy (*i.e.*, faster and more virtual CPUs), memory-heavy (more RAM), or storage-heavy workloads (fast instance storage). Special instance types are designated for specialized applications areas such as graphics-intensive applications (high-performance Graphical Processing Unit (GPU) cluster) or fields such as genome research, where customizable hardware acceleration[14] is beneficial. Custom instance types offer configurable resources within certain resource-specific limits (*e.g.*, 1-64 vCPUs[15]). The concept of bursting instance types is orthogonal to the previously introduced families of instance types and describes instance types that are designed to operate at baseline performance and handle period short-term burst of high load at an increased peak performance level [LS15].

2.1.4 Virtualization

Cloud computing leverages virtualization technology to implement resource sharing and on-demand provisioning. In virtualization, a Virtual Machine Monitor (VMM), also known as hypervisor, creates an abstraction layer that exposes virtualized computing resources to the user in

[8] https://azure.microsoft.com/en-us/regions/
[9] https://aws.amazon.com/about-aws/global-infrastructure/
[10] https://cloud.google.com/about/locations/
[11] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html
[12] https://cloud.google.com/compute/docs/machine-types
[13] https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-sizes-specs
[14] https://aws.amazon.com/ec2/instance-types/f1/
[15] https://cloud.google.com/custom-machine-types/

the form of an isolated VM [Por16]. Popek and Goldberg [PG74] formalize the following three defining characteristics for a correct VMM: fidelity, safety, and performance. Fidelity requires the VM environment to behave essentially identical to physical hardware. Safety ensures that the VMM fully manages all hardware resources and VMs remain isolated. Performance demands for minimal virtualization overhead. Further, a single physical machine (*i.e.*, host) can accommodate multiple VMs, which allows for resource sharing while maintaining isolation because of the safety characteristic. Additionally, the VMM provides a management interface to automatically create VMs, which is a key feature to deliver on-demand VM provisioning in the cloud.

The most common virtualization techniques in the cloud are Para-Virtualization (PV) and Hardware-assisted Virtual Machine (HVM). PV instances require VMM-aware guest operating system extensions, which are typically provided in the form of officially maintained cloud images[16] to initialize a VM. These guest extensions allow to very efficiently share hardware resources between different VMs [BDF+03] and replace privileged instructions with hypercalls to the VMM. HVM instances require hardware virtualization extension from the host CPU (*i.e.*, Intel VT or AMD-V) to improve hardware emulation performance. Therefore, customized guest operating systems are optional for fully virtualized guests[17]. However, custom PV extensions can still be used to improve performance of slowly emulated operations such as I/O. Two prominent open source VMMs that support these techniques are Xen [BDF+03] and Kernel-based Virtual Machine (KVM) [KKL+07], which is part of the Linux kernel since 2007[18].

2.2 Systems Performance Benchmarking

Systems performance benchmarking is the process of systematically evaluating the speed of computing resources, such as CPU or RAM, at the operating systems level.

2.2.1 Terminology

This section defines fundamental performance testing terminology.

System Under Test (SUT). The System Under Test (SUT) refers to the component or environment that is evaluated according to clearly defined metrics such as response time. In the context of IaaS cloud service performance evaluation, the VM obtained from the cloud provider constitutes the SUT.

Workload. Test workloads refer to the stimulation that is applied to the SUT. Real workloads are traced from actual load in production systems whereas synthetic workloads are artificially constructed. In the context of cloud computing, scale-out workloads [FAK+12] are inherently designed for cloud environments and their applications include mechanisms to dynamically acquire and release cloud resources during workload execution.

2.2.2 Micro Benchmarking

Micro benchmarking refers to the process of stimulating the SUT with simple artificial workloads (*i.e.*, micro benchmark) and measuring the performance of these operations. Micro benchmarks

[16] https://cloud-images.ubuntu.com/
[17] https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview#Guest_Types
[18] https://kernelnewbies.org/Linux_2_6_20#head-bca4fe7ffe454321118a470387c2be543ee51754

often test a very specific resource, such as CPU integer operations, and are used to identify bottlenecks. In contrast to application benchmarking, micro benchmarking is less sophisticated to conduct and interpret [Gre13]. Li *et al.* [LOZC12] compiled an extensive catalogue of metrics and links them predominantly to micro benchmarks, which can be used to obtain measurements for these metrics. Scheuner [Sch14] linked micro benchmarks in different resource categories to their usage in cloud performance evaluation studies.

2.2.3 Web Application Benchmarking

Web application benchmarks use an external load generator to run a workload against the Web application under test. The workload is specified in a test plan and executed by load testing tools, such as Apache JMeter[19]. Hereby, the load testing tool issues HTTP requests to the Web application and measures the response time of each individual request. Further metrics such as throughput (*i.e.*, number of requests per second) or failure rate are calculated to assess the Web application performance.

2.3 Cloud Benchmarking Automation

Benchmarking cloud environments is an elaborate and error-prone task and requires automation to minimize manual labor and prevent human error. Therefore, research and industry have proposed several tools to automate the process of repeatedly executing benchmarks in the cloud. The two most comprehensive and actively maintained tools backed by industry are CloudBench [SHG$^+$13] and Google's Perfkit Benchmarker[20]. Both command line tools are inherently designed to support scale-out workloads, include a comprehensive set of benchmarks, and support several cloud providers. CloudBench has been integrated into the standardized benchmark SPEC Cloud IaaS 2016[21] and the Perfkit Benchmarker is used in production to detect performance regressions[22]. Research has proposed approaches that are based on templated code generation [JSM$^+$12, JKC$^+$13], declarative Domain Specific Languages (DSLs) [CMS13], and Infrastructure as Code (IaC) benchmark provisioning [SLCG14, SCLG15]. The following section introduces the IaC-based CWB tool, which automates the benchmarks in this thesis.

2.3.1 Cloud WorkBench (CWB)

Cloud WorkBench (CWB) [SLCG14, SCLG15] is a Web-based cloud benchmark manager, which schedules and executes benchmarks without manual interaction. It fosters the definition of configurable and reusable CWB benchmarks that are entirely defined by means of code by leveraging IaC. Therefore, CWB benchmarks are portable across cloud providers and their regions with minimal effort.

[19] http://jmeter.apache.org/
[20] https://github.com/GoogleCloudPlatform/PerfKitBenchmarker
[21] https://spec.org/cloud_iaas2016/
[22] https://drive.google.com/file/d/0B66A4foojMJtRzRoYjZDRXNDQnc/view

Chapter 3

Related Work

This chapter presents related work in the areas of cloud performance variability, micro benchmarking, application benchmarking, and cloud instance type selection.

3.1 Cloud Performance Variability

Studies have extensively analyzed the stability of performance delivered by cloud providers. Thereby, hardware heterogeneity has received most attention and has been attributed to cause substantial variability within instance types. Although this field has become less active since reaching its peak between 2010 and 2013, continuous re-evaluation keeps cloud performance variability an ongoing topic in cloud computing research. Further, unpredictable performance is often a threat to internal validity for experiments conducted in cloud environments and therefore important to quantify.

One of the first large-scale studies to address variability in a cloud environment was conducted by Schad *et al.* [SDQR10]. They primarily use micro benchmarks to analyze variability over time within an instance, between instances of the same type, between data centers within the same region, and between data centers in different regions. They collected hourly measurements for over one month and revealed large variability around 20% for CPU, I/O, and network performance. The case study with a MapReduce workload validated the significant variability observed in a cloud environment compared to a local cluster. Other studies also observed high variability for instances of the same type [DPC10,LYKZ10] and identified hardware heterogeneity (*i.e.*, instances of the same type obtain different hardware; most importantly different CPU models) as major cause for CPU performance variability [CGPS12] beyond CPU sharing and noise due to multi-tenancy [BS10, WN10]. Hardware heterogeneity was exploited in so called *placement gaming* approaches where bad performing instances are discarded and well performing instances are kept to improve overall performance by 5% to 30% [FJV$^+$12] and reduce costs by up to 30% [OZL$^+$13, OZN$^+$12]. Further studies over time [Kot14, LC16] have shown that performance variability remained relevant, in particular for smaller instance types. The first long-term study over the course of one year [IYE11] discovered yearly and daily patters for some cloud services but also showed that most services perform particularly stable over certain periods.

3.2 Micro Benchmarking

A large body of work aims at measuring cloud service performance for individual resources such as CPU, I/O, memory, and network. One of the earliest studies in this field focused on benchmarking Amazon EC2 for high performance scientific computing [Wal08]. Subsequent work extends the scope by including more cloud services and led to some of the most important contributions in this field [OIY+10, IOY+11]. Assessing and comparing the performance of cloud services has also become a business and companies such as CloudHarmony[1] or Cloud Spectator[2] offer comparison services and publish their own price-performance analysis reports [Spe17]. Some latest work [VSTB16] investigates how to leverage container technology to obtain results in near real-time compared to much slower traditional VM-based approaches.

3.3 Application Benchmarking

Seeking for representative workloads for benchmarking cloud services has been an active field of research since the emergence of cloud computing. One of the earliest efforts geared towards more modern workloads for the cloud comprises the Cloudstone benchmark [SSS+08], which proposes a new interaction-heavy Web 2.0 workload. Several conceptual contributions [BKKL09, FAS+13] suggest ideas and guidelines on how to design and implement application benchmarks for cloud environments. The most important contribution in this field introduces an entire collection of scale-out workloads called CloudSuite [FAK+12]. For cloud databases, the extensible Yahoo! Cloud Serving Benchmark (YCSB)[3] maintains a large collection of scale-out workloads for database systems such as HBase[4], MongoDB[5], or Google Bigtable [CDG+08]. Further, the widely recognized SPEC consortium[6] published the benchmark suite called *SPEC Cloud^{TM} IaaS 2016* [Con16], which is specifically aimed to measure IaaS cloud performance.

The two largest challenges in this field are supposedly finding representative workloads for real-world use cases and ensuring reproducibility of benchmark execution. Seeking for representative workloads is a field that requires ongoing research attention because of continuous changes in technology and user behavior. Personal experience has shown that lots of application benchmarking research is almost impossible to reproduce due lack of documentation, discontinued software[7], closed source implementation [DPC10], or unavailable resources [SSS+08, ACC+02]. Nowadays, technologies are available to define reproducible application benchmarks and have been adopted for example in the latest version of CloudSuite[8] (3.0 as of May, 2017), which provides improved benchmarks and adopts Docker[9] container technology to facilitate deployment and increase transparency [PSF16].

3.4 Cloud Instance Type Selection

This section presents related work with the goal to guide cloud instance type selection by profiling and predicting application performance.

3.4.1 Application Performance Profiling

Application profiling research aims to capture the performance behavior of applications on different platforms and is most closely related to the work in this thesis.

[1] https://cloudharmony.com/
[2] http://cloudspectator.com/
[3] https://github.com/brianfrankcooper/YCSB
[4] http://hbase.apache.org/
[5] https://www.mongodb.com/
[6] https://www.spec.org/consortium/
[7] http://incubator.apache.org/projects/olio.html
[8] http://cloudsuite.ch/
[9] https://www.docker.com/

Evangelinou *et al.* [ECA+16] use system-level resource monitoring tools (*e.g.*, Pidstat[10]) to obtain a performance footprint consisting of 23 features from Java applications, cloud databases, and file I/O-intense applications. These generic application benchmarks are then used to evaluate the approach with an HTTP application whose performance footprint is mapped to predefined application categories. This result is combined with the Palladio design model [BKR09] and a Layered Queuing Network (LQN) performance model [RS95] to find optimal deployment options regarding combined service efficiency, which is a metric taking into account workload, cost, and performance. The work of this thesis differs by using more lightweight micro benchmarks instead of generic application benchmarks for application profiling and by providing direct performance estimates for each instance type instead of solely identifying the optimal instance type.

Canuto *et al.* [CBMG16] combine system-level resource monitoring with training micro benchmarks to predict power consumption in heterogeneous data centers. Their approach captures non-linear relations by applying polynomial or logarithmic transformations where necessary. The work of this thesis follows a similar workflow but uses micro benchmarks to estimate application performance instead of monitoring data to predict power consumption.

Hoste *et al.* [HPE+06] analyze program similarity at a compiler-level to rank the performance of different platforms. Application profiles are built from 47 microarchitecture-independent characteristics (*e.g.*, instruction mix including the percentage of integer operations). These profiles are then related to standardized micro benchmarks (*i.e.*, the SPEC CPU2000 benchmarks[11]) with similar characteristics. The three approaches normalization, principal component analysis, and genetic algorithms are applied to predict platform rankings and evaluated using the Spearman rank correlation coefficient. Their results reveal that small differences in the rank coefficient can translate to large differences in relative performance. This motivates the need for enhanced insights during instance type selection beyond single instance type recommendations or ordinal scale rankings (*e.g.*, cost and performance rankings for scientific applications [VAM+16]). Similar to the work in this thesis, Hoste *et al.* [HPE+06] relate an application profile to the performance levels of micro benchmarks. However, their work compares many different real hardware systems outside the context of cloud computing.

3.4.2 Application Performance Prediction

Predicting application performance in cloud and non-cloud environments is a broad field of research and closely related to the estimation model in this thesis.

Li *et al.* [LZZ+11, LZK+11] propose the CloudProphet tool, which collects resource traces of on-premise Web applications and replays them in cloud environments to accurately predict application performance for cloud instance types. During resource tracing, low-level system events for CPU, storage (disk and database), network, and locks are captured. Their proposed dependency extraction algorithm identifies causalities of event chains (*e.g.*, an incoming HTTP request triggers a block of I/O events). These dependency-annotated traces are replayed in the cloud against a migrated production database to obtain response time predictions with low error rates below 10% in most cases. They further demonstrate that dependency extraction is crucial for accurate predictions and that tracing overhead has low impact on application performance [LZZ+11]. In comparison, CloudProphet focuses on accurate predictions for few instance types whereas this thesis provides rough estimates for many different instance types. Hence, these approaches are complementary and could be combined to achieve broad instance type coverage and leverage CloudProphet to reduce the sampling effort, which is required to train the model in this thesis.

Alipourfard *et al.* [ALC+17] introduce the CherryPick system, which guides cloud configuration choices and iteratively refines runtime and cost predictions for distributed big data analytic

[10] `https://linux.die.net/man/1/pidstat`
[11] `https://www.spec.org/cpu2000/`

jobs using a Bayesian Optimization [BCDF10] model. Beyond instance type-dependent variables, this work also includes the number of VMs for the multi-machine workloads in the optimization model. The work in this thesis needs less initial training samples and covers other application domains with scientific computing and Web serving compared to data analytics.

Stewart and Shen [SS05] contribute a comprehensive performance model that claims to accurately predict the throughput and response time of multi-component online services by combining queuing models with system-level resource monitoring. They model per-component resource consumption and inter-component communication patterns as functions of input workload properties. The results reveal that remote method invocation overhead is a critical factor to achieve low error rates below 14%. In contrast to the work in this thesis, their applications are distributed across multiple instances. However, their evaluation is conducted in a local 20-node cluster with three different server types and does not consider a broad range of cloud instance types.

Chapter 4

Methodology

This chapter describes the methodology used to conduct the cloud benchmarking study. At the beginning, the process overview is outlined and in the following, each step of the process is described in detail. Finally, the threats to validity of the presented methodology are discussed.

4.1 Process Overview

Flowchart 4.1 summarizes the four-step benchmarking process including the input and output for each individual step. Firstly, in the *benchmark design* (4.2) step, benchmarks were selected, designed, and integrated into CWB. Hereby, CWB provides guidelines how to structure CWB-integrated benchmarks[1] and cloud benchmarking literature gives general guidelines on benchmark design and execution plans. The outcome comprises a set of CWB benchmarks that is automatically executable via a CWB server. Secondly, in the *benchmark execution* (4.3) step, these CWB benchmarks are repeatedly executed in a cloud environment via a CWB schedule. Thereby, the CWB server automatically collects performance measurements of the benchmark executions. Thirdly, in the *data pre-processing* (4.4) step, these measurements are imported, cleaned, and prepared for the main data analysis. Finally, the main *data analyses* are guided by the RQs introduced in Section 1.1 and lead to the results presented in Chapter 5.

Figure 4.2 illustrates the high-level architecture of this cloud benchmarking methodology. As a *Benchmark Manager*, the CWB Server coordinates the entire lifecyle of all CWB benchmark executions. Its *Scheduler* component triggers new executions and its *Cloud Manager* component abstracts the cloud Provider APIs, Cloud VM provisioning, and communication with the Cloud VM. Via the *Provider API*, *Cloud VMs*, which represent the SUT, are acquired. Within the cloud VM, the *Chef Client* controls the VM provisioning and the *CWB Client* steers the execution of the entire benchmark collection. The Chef Client fetches the provisioning configuration for the Cloud VM from the *Provisioning Service* and applies it to install and configure all *Micro* and *Application (App)* benchmarks. The CWB Client directs the execution order and handles communication with the CWB Server such as submitting result metrics to the Representational State Transfer (REST) API. Multi-VM benchmarks, such as iperf and WordpressBench (WPBench), submit their CWB tasks to the *Load Generator*, which generates the specified task workload from another dedicated cloud VM.

[1] `https://github.com/sealuzh/cwb-benchmarks#write-your-first-benchmark-getting-started`

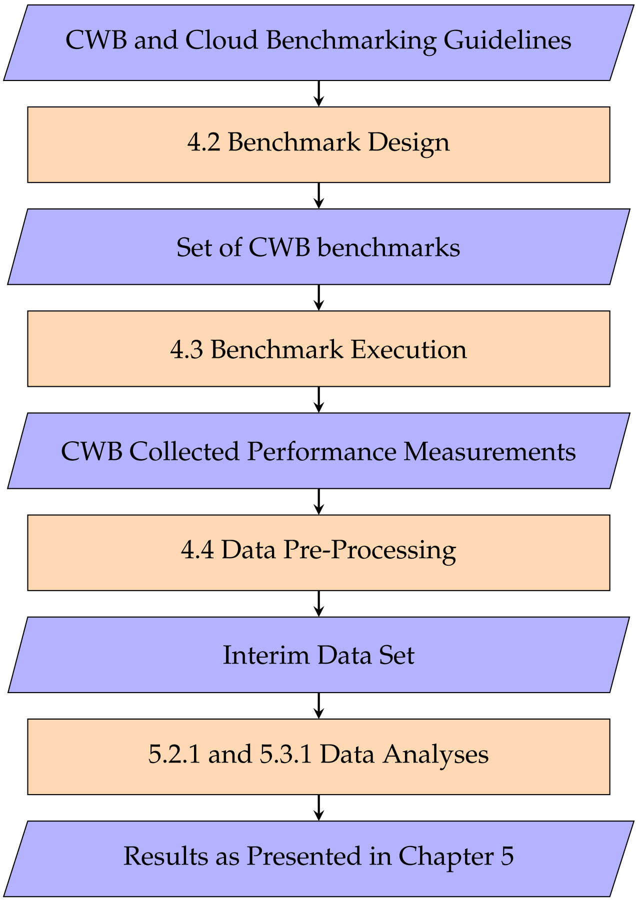

Figure 4.1: Process Overview Flowchart

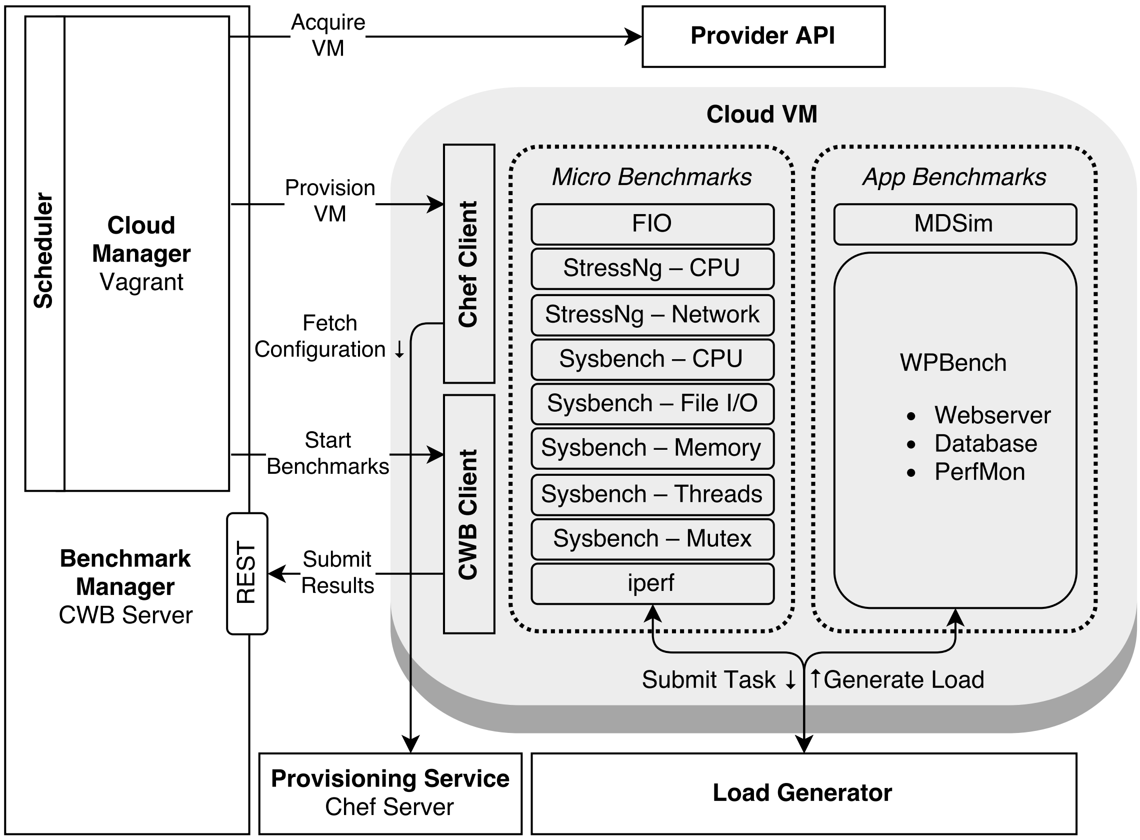

Figure 4.2: Architecture Overview

4.2 Benchmark Design

This section covers the extensions made to CWB, the integration of several micro benchmarks into CWB, and dedicates the last two subsections to the application benchmarks Molecular Dynamics Simulation (MDSim) and WPBench.

4.2.1 Cloud WorkBench

CWB was extended to modularly define benchmark plugins and combine them into a collection of benchmarks called benchmark suite. These extensions are then leveraged to package micro and application benchmarks into a combined CWB benchmark and implement a remote load generator to support multi-instance benchmarks.

Benchmark Plugins

The initial version of CWB [Sch14] had to be extended to support sequential execution of multiple benchmarks. Therefore, the monolithic script style of defining single benchmarks was replaced with a modular Object Oriented (OO) benchmark plugin system. Custom Chef extensions in combination with naming conventions[2] make modular benchmarks easily pluggable into a larger collection of benchmarks. This allows to execute multiple benchmarks in succession which is required to combine micro and application benchmarks. In addition, benchmarks can now be defined much more concisely and elegantly, individual functionality can be unit-tested using Rspec[3], and smoke-tested locally using the *cwb* command line utility[4].

A CWB benchmark plugin typically executes a benchmark command, extracts some metrics of interests from the result, and submits these metrics to the CWB server. Each benchmark plugin overrides the Ruby *execute* hook method. Therein, the Linux command to start the benchmark tool is composed and executed. After completion of the benchmark execution, result metrics from the standard output stream are extracted via regular expression pattern matching. The *submit_metric* method from the CWB client library[5] is then used to submit named and timestamped metrics to the CWB server.

Benchmark Suite

Benchmark suites were introduced to CWB to control the execution order of a collection of CWB benchmark plugins. The OO design of the CWB benchmark execution systems allows to define benchmark suite[6] subclasses that control the execution of the entire collection of benchmarks. In addition, cross-cutting concerns can be handled in such benchmark suites such as logging execution progress, notifying the CWB server, handling execution errors, or reporting metadata.

Randomized Multiple Interleaved Trials (RMIT). Abedi and Brecht [AB17] reveal considerable flaws in the methodoloy used by many performance studies conducted in cloud environments. Simulations with performance traces from previous benchmarking experiments [SDQR10] have shown that inappropriate ordering of benchmark executions "could lead to erroneous conclusions" [AB17]. The *Single Trial* approach, where every benchmark is executed only once, negelects

[2] `https://github.com/sealuzh/cwb-benchmarks/tree/master/cwb#resource`
[3] `http://rspec.info/`
[4] `https://github.com/sealuzh/cwb-benchmarks#local-testing`
[5] `http://www.rubydoc.info/gems/cwb/Cwb/Client`
[6] `https://github.com/sealuzh/cwb/blob/master/lib/cwb/benchmark_suite.rb`

intra-instance variability. The *Multiple Consecutive Trials (MCT)* approach, where every benchmark is repeated N times before proceeding with the next benchmark, fails to take environmental changes into account. The *Multiple Interleaved Trials (MIT)* approach, where in a first round every benchmark is executed once followed by N repetitions of this first round, ignores periodic patterns that could cause performance deviations for particular repetitions. Therefore, the authors recommend the use of the *Randomized Multiple Interleaved Trials (RMIT)* approach for fair comparison of competing alternatives. The RMIT approach is a variation of the MIT approach where the benchmark order within the individual rounds is randomized instead of kept constant. The cloud benchmarking experiments conducted in this thesis follow the RMIT methodology, which is implemented as a CWB benchmark suite[7].

RMIT Benchmark Suite. Beyond implementing the RMIT methodology, the RMIT benchmark suite logs execution progress and reports metadata from the instance (*e.g.*, CPU model name), system (*e.g.*, gcc compiler version), and individual benchmarks (*e.g.*, version number). Execution progress is logged by submitting a timestamped `START` and `END` metric around executing every single benchmark plugin. Additionally, the benchmark order according to the RMIT methodology is reported for every CWB execution The instance metadata metrics are explained in Table 4.1 and the benchmark version numbers are given in Table 4.2.

Metric Name	**Explanation**
CPU model	Processor type (*e.g.*, *Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz)*
CPU cores	Number of total CPU cores as revealed to the VM
RAM total	Exact amount of memory in KB available
Compiler version	Version string (*e.g.*, *gcc (Ubuntu 4.8.4-2ubuntu1 14.04.3) 4.8.4*)

Table 4.1: Instance Metadata Metrics

RMIT Combined Benchmark

The entirety of benchmarks used in this thesis are packaged together within the RMIT combined benchmark. This CWB benchmark bundles the RMIT benchmark suite together with the benchmark plugins of all micro benchmarks (4.2.2) and the application benchmark MDSim (4.2.3) within a single Chef cookbook called *rmit-combined*[8]. The application benchmark WPBench is specified as internal dependency and automatically resolved by the Berkshelf[9] dependency manager.

Load Generator

The load generator provides a REST endpoint to submit CWB tasks, which can run arbitrary workload against the SUT. These CWB tasks follow the guidelines of a CWB benchmark plugin and thus provide an instance-independent execution environment for benchmark plugins. This allows to define load generating benchmark plugins for multi-machine benchmarks (*e.g.*, iperf

[7] https://github.com/sealuzh/cwb-benchmarks/blob/master/rmit-combined/files/default/rmit_benchmark_suite.rb#L72

[8] https://github.com/sealuzh/cwb-benchmarks/blob/master/rmit-combined/

[9] https://docs.chef.io/berkshelf.html

and WPBench) and run their workloads (*i.e.*, Transmission Control Protocol (TCP) network test and JMeter test plan) against the SUT. The load generator is implemented as Ruby on Rails[10] application and available as open source software on Github[11]. It can be automatically deployed on a dedicated instance using Vagrant[12] and Chef[13].

4.2.2 Micro Benchmarks

The selection of micro benchmarks aims for broad-resource coverage and specific-resource testing while trying to minimize redundancy and profiling execution time. To obtain an extensive instance profile, the selected micro benchmarks cover resources in the domains computation, I/O, network, and memory. Within each category, micro benchmarks were selected to specifically test different aspects. For example, the I/O domain is divided into low-level disk I/O and higher-level file I/O. Each of these subdomains, can be further divided based on operation type (*e.g.*, sequential/random and read/write) or operation size (*e.g.*, 4k/8k block size). Given the large space of micro benchmarks, benchmark selection tries to avoid very similar benchmarks that are expected to deliver redundant information and also attempts to tune execution time under the premise that still meaningful results are delivered. Consequently, exceedingly long running benchmarks without suitable tuning options had to be discarded. An additional practical criteria was to favor benchmarks from the same benchmarking tool where suitable to avoid unnecessary installation effort.

The selected micro benchmarks are integrated into CWB using CWB benchmark plugins. The CWB integration basically follows the description in Subsection 4.2.1 whereby deviations are described in the following for each micro benchmark. Additionally, Table 4.2 lists the micro benchmark tools and their version numbers used in this thesis.

Benchmark Tool	**Version**	**Source**
Flexible I/O (FIO) Tester	2.1.10	[14]
iperf	2.0.5 (pthreads)	[15]
StressNg	0.07.27	[16]
Sysbench	0.4.12	[17]

Table 4.2: Micro Benchmark Tools

Flexible I/O Tester (FIO)

The *FIO* benchmark tests sequential write (*fio/4k-seq-write*) and random read (*fio/8k-rand-read*) disk I/O performance. It uses the highly configurable FIO tool[18] as an I/O workload simulator. Immediately after each execution, the generated temporary files are deleted to set the system into pristine state and avoid running out of storage.

[10] `http://rubyonrails.org/`
[11] `https://github.com/joe4dev/load-generator`
[12] `https://www.vagrantup.com/`
[13] `https://www.chef.io/`
[18] `https://github.com/axboe/fio`

fio/4k-seq-write. To assess sequential disk write performance, this benchmark replicates the study setup from [SLCG14] using the same software and benchmark settings. Listing 4.1 reports the shell command and options used to execute the benchmark. The sequential write workload size is set to 1 Gibibyte (GiB) using the default block size of 4 KiB (4096 bytes). Furthermore, the direct I/O mode ignores caches to test raw write performance and the refill buffers mode circumvents SSD compression effects.

```
fio --name=write --numjobs=1 --ioengine=sync --rw=write --bs=4k \
    --size=1g --direct=1 --refill_buffers=1 --filename=fio.tmp
```

Listing 4.1: FIO 4k Sequential Write Shell Command

fio/8k-rand-read. To assess random read performance, this benchmark setup is guided by the recommendations[19] from a well-known cloud computing performance engineer[20]. The shell command in Listing 4.2 aims to simulate typical file access read operations using a non-uniform access distribution for a mixed I/O and cache test. The benchmark runtime is limited to 60 seconds and the block size is set to 8 KiB (8192 bytes).

```
fio --time_based --runtime=60 --clocksource=clock_gettime \
    --numjobs=1 --name=randread  --rw=randread --bs=8k --size=2g \
    --random_distribution=pareto:0.9 --filename=fio.tmp
```

Listing 4.2: FIO 8k Random Read Shell Command

Both I/O scenarios extract and report the 6 metrics listed in Table 4.3. The benchmark duration is reported for every micro benchmark to quantify the profiling effort. Notice that bandwidth and Input/Output Operations per Second (IOPS) are redundant metrics but are both reported for convenience because bandwidth is rather used for sequential I/O operations whereas IOPS is oftentimes preferred for random I/O operations. Latency is captured to assess the efficiency of the cloud storage connectivity to the compute instances. Additionally, the 95th latency percentile attributes for typical long-tail distributions observed in cloud environments [SDQR10, XMNB13]. Finally, disk utilization serves as control metric whether the benchmark actually saturates disk I/O to the expected extent.

iperf

The *iperf* benchmark is used to measure intra-cloud TCP network bandwidth between the cloud VM (CWB iperf and iperf server) and a dedicated load generator. This multi-machine benchmark integrates substantially different into CWB compared to all other single-machine micro benchmarks. Figure 4.3 illustrates how an iperf benchmark execution is integrated into CWB. Within the context of the iperf CWB benchmark plugin, *CWB iperf* starts the daemonized iperf server. Subsequently, CWB iperf submits the iperf task to a *Load Generator*, which is hosted on a dedicated cloud VM, and waits for a completion message. In the meantime, the load generator sequentially executes the single- and multi-thread scenario against the iperf server. Hereby, the resulting met-

[19] http://dtrace.org/blogs/brendan/2014/01/10/benchmarking-the-cloud/
[20] https://twitter.com/brendangregg

Metric Name	Unit	Explanation
Duration	ms	Total time it takes to execute the read/write I/O workload
Bandwidth	KiB/s	Average read/write speed
IOPS	Ops/s	Average number of read/write operations performed per second
Latency	μs	Average time it takes until an issued I/O request is handled
Latency 95th Percentile	μs	Upper bound wherein 95% of the I/O requests are handled
Disk Utilization	%	Percentage of time where the disk is busy

Table 4.3: FIO Metrics

rics are submitted to the CWB server immediatley after execution. Finally, the load generator notifies CWB iperf to stop the iperf server.

Listing 4.3 reports the iperf command with its static and dynamic options. The static options specify the 30 seconds runtime and increase the default (4 KB) buffer size to 128 KB to test maxium bandwidth[21]. The dynamic options configure the client/server connection and distinguish between the single- and multi-threaded scenario. The load generator, in client mode (`-c`), connects to the iperf server using the dynamically resolved `$HOST` variable, which points to the private Internet Protocol (IP) address of the iperf server. For the single-thread scenario, the "number of parallel client threads to run"[22] is set to `1` via the `$NUM_CPU_CORES` variable whereas for the multi-thread scenario, the `$NUM_CPU_CORES` variable is substituted with the number of virtual cores available to the cloud VM. These two scenarios are tested to investigate whether a single connection is sufficient for reaching maximum network performance or a multi-threaded workload is able to exceed this baseline.

```
iperf -c $HOST -l 128k -t 30 -P $NUM_CPU_CORES
```

Listing 4.3: iperf Shell Command

For both thread scenarios, the iperf benchmark reports the 2 metrics listed in Table 4.4. Duration is solely reported for consistency reason across all benchmarks because iperf is statically configured to always run for 30 seconds. Bandwidth is the metric of interest, which measures the average throughput of the TCP network from client to server.

Metric Name	Unit	Explanation
Duration	s	Total time it takes to execute the iperf workload
Bandwidth	Mbits/s	Average TCP network speed

Table 4.4: iperf Metrics

[21] http://dtrace.org/blogs/brendan/2014/01/10/benchmarking-the-cloud/
[22] http://manpages.ubuntu.com/manpages/precise/man1/iperf.1.html

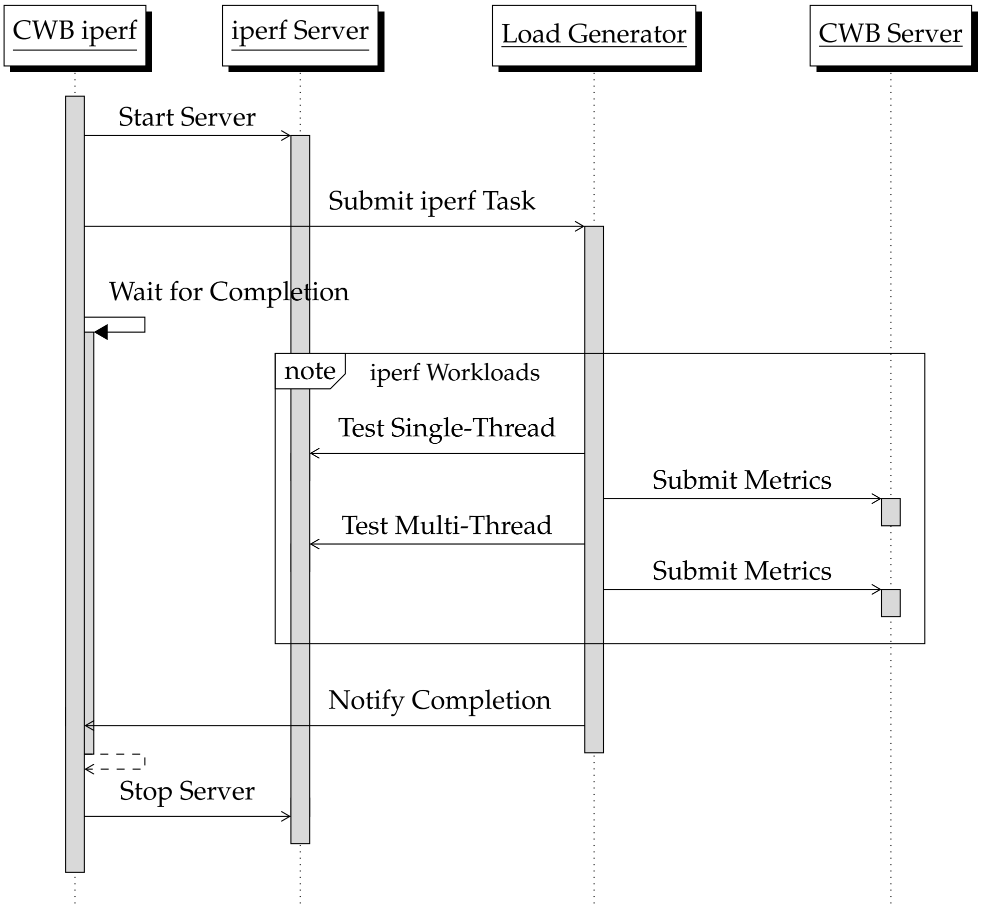

Figure 4.3: iperf Benchmark Execution

StressNg – CPU

The StressNg benchmark tool[23] contains over 170 stress tests (*i.e.*, stressors) and is designed to exercise various specific physical and operating system resources. From the almost 70 CPU specific stressors, 8 stressors were selected to cover the following 3 domains: data types (integer, double), language primitives (loops, recursive function calls), and algorithms (Euler, Fibonacci, matrix product). These stressors, referenced as `$STRESSOR`, are executed using the shell command shown in Listing 4.4. All stressors assess single thread performance and run for 10 seconds to minimize profiling effort.

```
stress-ng --cpu 1 --cpu-method $STRESSOR -t 10 --metrics-brief
```

Listing 4.4: StressNg – CPU Shell Command

Table 4.5 lists the 2 metrics captured for each of the 8 stressors. Beyond the default duration metric, throughput estimates the performance of a stressor by capturing its iteration count. Thus, this "bogus operations per second" counter is a relative metric and cannot be compared across different stressors. The StressNg documentation[24] also indicates its insufficient scientific accuarcy as a benchmarking metric. However, for inter-instance performance estimates, StressNg can still deliver useful information and is also used for such a profiling purpose in [CBMG16].

Metric Name	Unit	Explanation
Duration	s	Total time it takes to execute the StressNg CPU workload
Throughput	bogo ops / s	Number of iterations achieved by the stressor

Table 4.5: StressNg Metrics

StressNg – Network

The StressNg network benchmark tests local network performance. Listing 4.5 shows the shell command used to run 4 selected stressors from the StressNg network class workload. The selected stressors test socket operations (sockfd, epoll), User Datagram Protocol (UDP) operations (udp) and Internet Control Message Protocol (ICMP) random ping flooding (icmp-flood). Notice that the ICMP stressor requires root permission and an exclude list had to be used because StressNg provides no include option. Equivalently to the StressNg CPU benchmark, these stressors are sequentially executed running for 10 seconds each. The total duration is reported once for the entire class of network stressors instead of individually for each stressor. Otherwise, the throughput metric works as described for the StressNg CPU benchmark (*cf.*, Table 4.5).

Sysbench – CPU

Sysbench[25] is a benchmark suite with micro benchmark workloads for CPU, file I/O, memory, threads, and mutexes. The additional Online Transaction Processing (OLTP) database workload

[23] `http://kernel.ubuntu.com/~cking/stress-ng/`
[24] `https://wiki.ubuntu.com/Kernel/Reference/stress-ng`
[25] `https://github.com/akopytov/sysbench`

```
sudo stress-ng --sequential 1 --class network -t 10 --metrics-brief \
               --exclude dccp,sctp,sock,sockpair,udp-flood
```

Listing 4.5: StressNg – Network Shell Command

was not considered for this study to avoid interference with the Wordpress benchmark described in Subsection 4.2.4.

Listing 4.6 reports the Sysbench CPU shell command used for single- and multi-thread performance measurements. This CPU workload computes the primality test in the interval $[3, 20000]$. The variable `$NUM_CPU_CORES` is substitued with 1 for the single-thread scenario and with the number of virtual cores available to the VM for the multi-thread scenario. These two scenarios were included to check CPU scalability[26] and identify any potential cloud limits[27]. For both thread scenarios, the Sysbench CPU benchmark reports its total duration as summarized in Table 4.6.

```
sysbench --test=cpu --cpu-max-prime=20000 \
         --num-threads=$NUM_CPU_CORES run
```

Listing 4.6: Sysbench – CPU Shell Command

Metric Name	Unit	Explanation
Duration	s	Total time it takes to check primality for all numbers in $[3, 20000]$

Table 4.6: Sysbench – CPU Metrics

Sysbench – File I/O

The Sysbench file I/O benchmark tests three different scenarios. All these scenarios use a dynamic workload size (*i.e.*, `${FILE_TOTAL_SIZE_GB}`) configured to be twice the amount of total instance RAM to obtain a more realistic cache to disk I/O ratio[28]. To avoid running out of disk space during the benchmark run, the disk space attached to VMs had to be increased for larger instance types with more RAM and the test files had to be cleaned up immediately after workload execution. The sequential write scenario (`seqwr`), as shown in Listing 4.7, uses a larger block size of 1 MB for increased throughput as recommended for sequential I/O tests[29]. Conversely, the random write scenario (`$MODE=rndwr`), as shown in Listing 4.8, uses a smaller block size of 4 KB as it is typical for randomly accessing small chunks. This scenario is mirrored with the same settings for the random read scenario (`$MODE=rndrd`). The read scenario slightly differs in benchmark execution because test files have to be laid out using the `prepare` parameter prior to running the workload via `run`.

[26] https://wiki.mikejung.biz/Sysbench#Sysbench_CPU_Tests
[27] http://dtrace.org/blogs/brendan/2014/01/10/benchmarking-the-cloud/
[28] https://wiki.mikejung.biz/Sysbench#Sysbench_Prepare
[29] https://wiki.mikejung.biz/Sysbench#Sysbench_Fileio_file-block-size

```
sysbench --test=fileio --file-total-size=${FILE_TOTAL_SIZE_GB}G \
         --file-block-size=1M --file-test-mode=seqwr run
```

Listing 4.7: Sysbench – File I/O Sequential Write Shell Command

```
sysbench --test=fileio --file-total-size=${FILE_TOTAL_SIZE_GB}G \
         --file-block-size=4K --file-test-mode=$MODE run
```

Listing 4.8: Sysbench – File I/O Random Write/Read Shell Command

Table 4.7 summarizes the metrics reported for each Sysbench file I/O scenario. These metrics have their corresponding lower-level FIO counterparts (*cf.*, Table 4.3) with different units and other terminology for throughput (*cf.*, bandwidth).

Metric Name	**Unit**	**Explanation**
Duration	s	Total time it takes to execute the I/O workload
Throughput	MB/s	Average read or write speed
Latency	ms	Average time it takes until an issued I/O request is handled
Latency 95th Percentile	ms	Upper bound wherein 95% of the acI/O requests are handled

Table 4.7: Sysbench – File I/O Metrics

Sysbench – Memory

The Sysbench memory benchmark tests 2 different scenarios of writing data into an allocated memory (*i.e.*, RAM) buffer. Listing 4.9 shows the shell command for the scenarios with default block size and larger block size. The default block size of 1 KB (`$BLOCK_SIZE=1K`) uses a 1 GB workload (`$TOTAL_SIZE=1G`). For the larger block size of 1 MB (`$BLOCK_SIZE=1M`), the workload was increased to 10 GB (`$TOTAL_SIZE=10G`) to partially compensate reduced execution time due to fewer iterations. Testing different workload sizes in the interval $[1, 1000]$ has shown that throughput does not differ meaningfully and therefore workload size was optimized to obtain faster execution time. For both of these scenarios the throughput and duration is reported as summarized in the metrics table 4.8.

```
sysbench --test=memory --memory-block-size=$BLOCK_SIZE \
         --memory-total-size=$TOTAL_SIZE run
```

Listing 4.9: Sysbench – Memory Shell Command

Sysbench – Mutex

The Sysbench mutex benchmark tests the speed of single-thread mutex lock operations. Listing 4.10 shows the configuration used to repeatedly request a single mutex lock within a loop.

Metric Name	Unit	Explanation
Duration	s	Total time it takes to execute the Sysbench – Memory workload
Throughput	MB / s	Sequential write speed to RAM

Table 4.8: Sysbench – Memory Metrics

The single metric of interest being reported for this benchmark is the total time it takes to acquire and release all $5 * 10^7$ mutex locks (Table 4.9).

```
sysbench --test=mutex --mutex-num=1 --mutex-locks=50000000 \
         --mutex-loops=1 run
```

Listing 4.10: Sysbench – Mutex Shell Command

Metric Name	Unit	Explanation
Duration	s	Total time it takes to execute the Sysbench – Mutex workload

Table 4.9: Sysbench – Mutex Metrics

Sysbench – Threads

The Sysbench threads benchmark simulates a single-thread and a highly concurrent thread lock scenario. Using the shell command in Listing 4.11, the variable `$NUM_THREADS` is set to `1` for the single-thread scenario and to `128` for the highly concurrent scenario where many threads compete for a single thread lock. In this workload, every thread acquires a lock, performs a yield operation to pause the current thread, and subsequently releases the lock when being rescheduled. The average time it takes to run such a single `lock-yield..unlock` sequence is reported as latency in addition to the total duration as listed in the metrics table 4.10. In comparison to the Sysbench mutex benchmark which focuses on single-thread mutex lock performance, this thread benchmark additionally investigates scheduler performance via the highly concurrent thread lock scenario.

```
sysbench --test=threads --thread-locks=1 --num-threads=$NUM_THREADS run
```

Listing 4.11: Sysbench – Threads Shell Command

4.2.3 Molecular Dynamics Simulation (MDSim)

The MDSim benchmark serves as a representative for scientific computing applications. An MDSim performs step-wise evolution of moving particles in a three-dimensional space according

Metric Name	Unit	Explanation
Duration	s	Total time it takes to execute the Sysbench – Threads workload
Latency	ms	Average time it takes to run a single `lock-yield..unlock` sequence

Table 4.10: Sysbench – Threads Metrics

to the physical laws considering particle positions and velocities [BCX+06]. This scientific application was also used for benchmarking cloud instances by Varghese *et al.* [VAM+14, VSTB16, VAM+16].

The MDSim benchmark integrates into CWB similar than a typical micro benchmark. MDSim ships as a single C source file and has to be compiled on the target system during the installation step. For compilation, the option `-fopenmp` and the `gcc` version `4.8.4` is used. Compared to the original version used in [VAM+16], the number of particles in the simulation and number of steps (*i.e.*, iterations) was exposed as a dynamic parameter to conveniently adjust the workload size. While maintaining the same number of simulation steps as in [VAM+16], the number of particles had to be reduced from 10000 to 1000 to reduce simulation time from multiple hours to below 10 minutes. This total simulation time is reported as the duration metric (Table 4.11).

Metric Name	Unit	Explanation
Duration	s	Total time it takes to simulate all 200 steps

Table 4.11: MDSim Metrics

4.2.4 Wordpress Benchmark

The Wordpress benchmark called *WPBench* was designed and implemented to serve as a representative for Web serving applications. WPBench runs different JMeter[30] load scenarios against a Wordpress[31] server and measures typical metrics such as response time and throughput. Wordpress was chosen because it is the most popular Content Management System (CMS) software (59% market share) used by 27.0% of the top 10 million websites[32] as of May 5, 2017 according to the Web technology surveys from W3Techs[33]. It has also been used for benchmarking cloud VMs [BLL+14].

Figure 4.4 illustrates the interaction design of WPBench. The overall interaction pattern resembles the iperf micro benchmark as described in section 4.2.2 (*cf.*, Figure 4.3). On the asynchronous *Start server* call, the Wordpress server starts the Web server, the corresponding database, and a performance monitoring agent. The *Submit JMeter task* message contains the JMeter test plan for all three load scenarios. These scenarios create detailed log files which are analyzed to summarize each test scenario. While the log files remain on the load generator for more detailed analysis, the metric summary is submitted to the CWB server. Afterwards, the Wordpress server notifies test

[30] `http://jmeter.apache.org/`
[31] `https://wordpress.org/`
[32] `https://w3techs.com/technologies`
[33] `https://w3techs.com/technologies/overview/content_management/all`

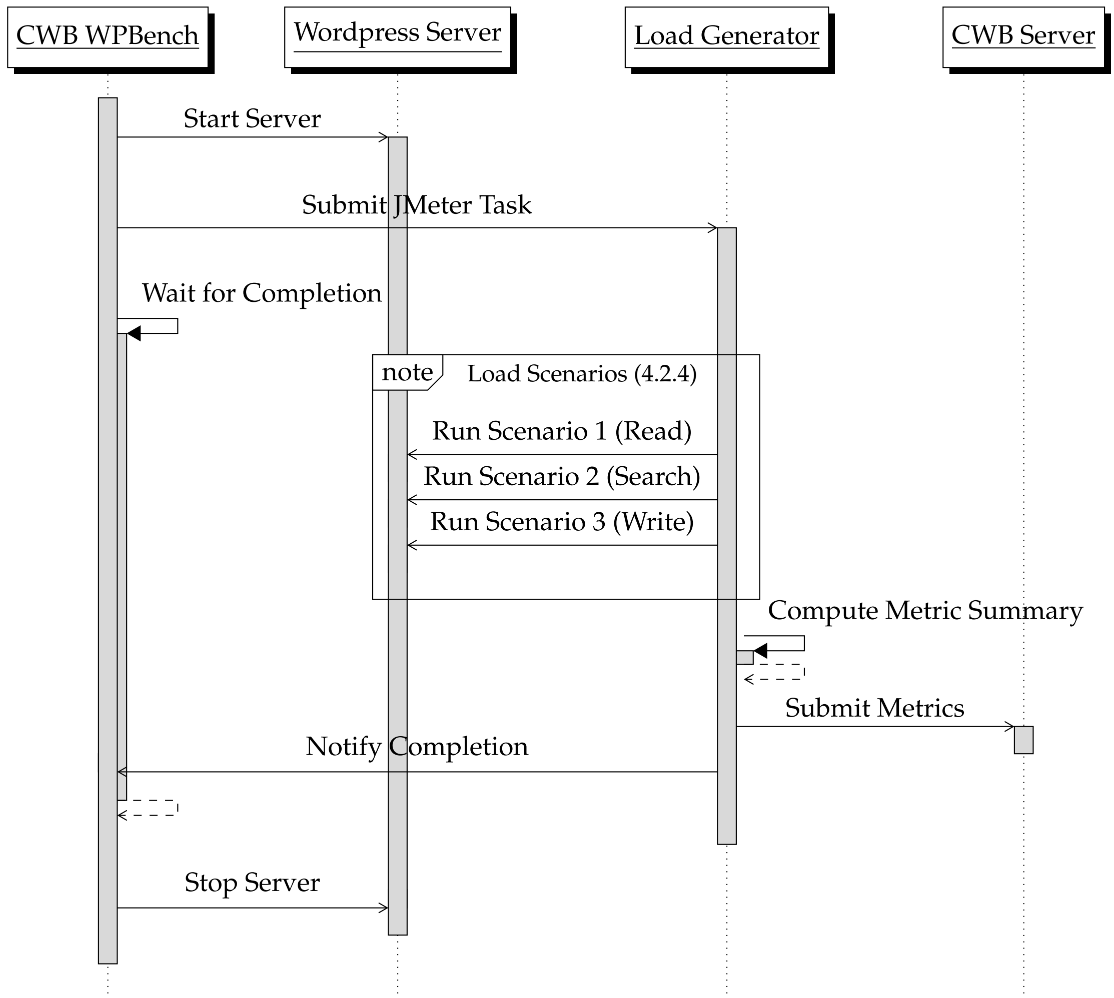

Figure 4.4: WPBench Execution

completion and CWB WPBench stops the server to prevent interference with subsequent benchmarks.

WPBench is substantially more involved than all other benchmarks used in this thesis. The following sections elaborate on the extended Wordpress installation automation, the generation of test data sets including migrations, the three different load scenarios, and the load patterns within these scenarios. Additionally, system resource monitoring during test execution and the distributed testing mode are described.

Automated Wordpress Installation

WPBench is able to automatically install and setup Wordpress including all of its dependencies to achieve portability across different platforms and cloud providers as encouraged by CWB [SLCG14]. The Wordpress installation builds upon the Chef cookbook *wordpress* from the Chef Supermarket community[34] to implement necessary extensions required for WPBench within a

[34] https://supermarket.chef.io/cookbooks/wordpress

fork[35]. Beyond several corrective changes, capabilities to automatically setup the Wordpress core and install plugins were added. Plugin support was mandatory for the *Fakerpress* plugin[36] to generate a test data set and for the *Disable Check Comment Flood* plugin to disable spam protection during load testing. The most relevant software packages are summarized in Table 4.12.

Software	**Version**	**Source**
Wordpress	4.7.1	[37]
Wordpress plugin – FakerPress	0.3.1	[38]
Wordpress plugin – Disable Check Comment Flood	1.0	[39]
PHP	5.5.9	[40]
MySQL	5.5.54	[41]

Table 4.12: Wordpress Installation – Software Packages

Test Data Set

A Wordpress test data set is generated leveraging the *Fakerpress* Wordpress plugin. Table 4.13 summarizes the quantities of the data set that comprises users in different roles, categories, tags, comments, and posts including sample images. FakerPress is configured to obtain real images from the 500px[42] photographer community. These images are stored with on-the-fly generated identifiers by Wordpress. Thus, the test plan has to be adjusted for each data set. Furthermore, data generation is too time consuming (~10-20 minutes) to perform on every new instance from scratch. For these reasons, the test data set is typically cached by creating a reusable VM image. This is the only part that comprises some manual work (*e.g.*, image capturing) and thus makes the benchmark not fully portable in an automated way. However, a migration script and instance cleanup script is provided by WPBench to minimize this one time effort.

Load Scenarios

The three different load scenarios of WPBench aim to simulate short read, search, and write Web browsing sessions. To accurately capture representative Web browser scenarios, a JMeter proxy[43] recorded these real Firefox browsing sessions. In iterative refinement, these captured traces were generalized, organized, and enriched with additional configurations. For generalization, all hard-coded Web server addresses had to be replaced with a dynamically configurable `site` variable. All external requests (*e.g.*, loading fonts from a Content Delivery Network (CDN) provider or avatar icon from Gravatar[44]) were disabled to prevent them from distorting the response times of the SUT. Finally, repetitive operations such as posting a comment had to be parametrized with dynamic content to mimic more representative workload and to circumvent the double posting validation. Organizing these over 200 HTTP requests required some logical grouping according to their inherent interaction structure (*e.g.*, group all immediate and dynamic HTTP request caused by a user search) to keep the test plan manageable. Additional configuration elements include

[35] https://github.com/joe4dev/wordpress
[36] https://wordpress.org/plugins/fakerpress/
[42] https://500px.com/
[43] https://www.blazemeter.com/blog/jmeters-superpower-http-proxy-server
[44] http://en.gravatar.com/

Component Type	Attribute	Quantity
Users	Administrator	10
Users	Editor	50
Users	Author	100
Users	Contributor	500
Users	Subscriber	1000
Taxonomies	Category	20
Taxonomies	Tags	50
Posts	Pages (100% image rate)	20
Posts	Normal (75% image rate)	200
Comments		2000

Table 4.13: WPBench Test Data Set

browser session managers, assertions, and different timers. The cookie manager simulates realistic handling of HTTP cookies. Assertions selectively check whether the Web server responses deliver the expected page content. Uniform random timers precede the first interaction of each scenario to prevent bursty request patterns (*i.e.*, all threads performing the same request at the same time). All subsequent interactions then use a constant timer to mimic users thinking time between interactions in the millisecond interval $[500, 4000]$.

Load Pattern

WPBench uses a step-wise growing load pattern configured as shown in Table 4.14 and visualized in Figure 4.5. Starting with 10 threads (*i.e.*, virtual users), every 30 seconds another 10 threads are added until after 5 minutes the target concurrency of 100 threads is reached. This load is then kept constant for 3 minutes. Subsequently, the tear-down phase runs until all 100 threads finished their current load scenario. This load pattern is implemented via the JMeter *Concurrency Thread Group* plugin[45], which is a more modern alternative than the default JMeter thread group[46].

Target concurrency	100	Ramp-up time	5 min
Ramp-up steps count	10	Hold target rate time	3 min

Table 4.14: WPBench Load Pattern Configuration

System Resource Monitoring

Following the active benchmarking methodology proposed in [Gre13], several resources of the SUT were monitored at system-level during test plan execution. Table 4.15 lists the monitored

[45] https://www.blazemeter.com/blog/advanced-load-testing-scenarios-jmeter-part-4-stepping-thread-group-and-concurrency-thread

[46] http://jmeter.apache.org/usermanual/test_plan.html#thread_group

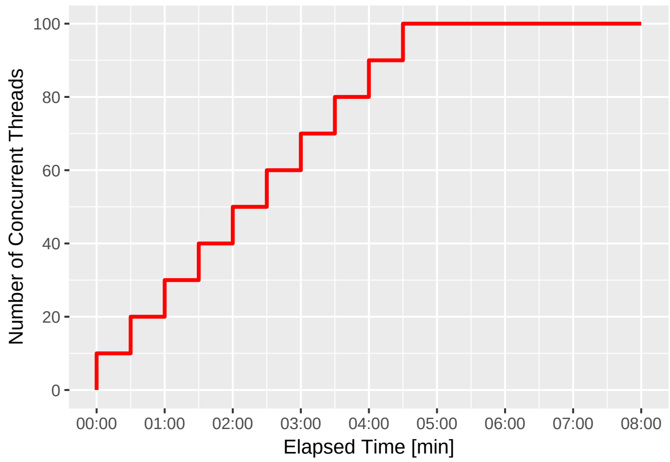

Figure 4.5: WPBench Load Pattern Visualization

Metric	Unit	Explanation
Memory utilization	%	Percentage of total instance RAM used
Disk I/O queue length	length	Number of outstanding I/O requests
Network I/O received	bytes	Number of bytes received
Established TCP connections	number	Number of established TCP connections
CPU combined (user + system)	%	Time where CPU is busy
CPU idle	%	Time where CPU is idle
CPU steal	%	Time where CPU is allocated to another tenant

Table 4.15: Monitored System-Level Resource Metrics

metrics[47] covering memory, disk I/O, network I/O, TCP connections, and three different CPU utilization indicators. Three CPU utilization metrics (*i.e.*, combined, idle, steal) were used to attribute for CPU throttling as discussed in [LS15] because cloud VMs are often artificially throttled by the VM hypervisor and thus do not get all CPU cycles. These three metrics sums up to 100% utilization together with the additional *iowait* metric. Monitoring is implemented using the JMeter plugin *PerfMon*[48]. The PerfMon server agent[49] gets automatically installed during the WPBench installation and started at the beginning of the WPBench execution.

Distributed Testing

A distributed testing mode was implemented to support powerful instance types where one single load generator is unable to generate sufficient workload. Using the JMeter remote testing mode[50], a load generator serves as a coordinating JMeter master node and N JMeter slave nodes

[47] https://jmeter-plugins.org/wiki/PerfMonMetrics/
[48] https://jmeter-plugins.org/wiki/PerfMon/
[49] https://jmeter-plugins.org/wiki/PerfMonAgent/
[50] http://jmeter.apache.org/usermanual/remote-test.html

concurrently run the load scenarios against the SUT. Supporting the distributed testing mode required a few adjustments to the test plan such as naming the thread groups dependent on the slave machine. This is necessary to distinguish the source of each response time sample in the log files.

4.3 Benchmark Execution

This section describes how the previously designed benchmarks are configured and subsequently automatically executed in CWB. It also describe where and how the resulting performance metrics are persisted.

The benchmark configuration in CWB defines the provider-specific resources, refers to the previously described RMIT combined benchmark (4.2.1), and specifies an execution schedule. Figure 4.6 depicts all these elements within the CWB Web interface. The provider-specific (*e.g.*, AWS) Vagrantfile section (*i.e.*, line 5-17) describes the geographic area (*i.e.*, region) of the data center (*e.g.*, *eu-west-1* in Ireland) and the isolated location (*i.e.*, availability zone[51]) within this region (*e.g.*, *eu-west-1a*). It also refers to a captured base image (*e.g.*, the Amazon Machine Image (AMI) containing the cached test data set described in 4.2.4). Furthermore, it specifies the instance type (*e.g.*, *m1.small*), a list of security groups (*e.g.*, *cwb-web* defining firewall rules to allow Secure Shell (SSH) and HTTP traffic), and the storage attached to the VM (*e.g.*, 12 GB *gp2* SSD EBS). The benchmark-specific Vagrantfile section (*i.e.*, line 19-31) refers to the adjusted test plan for the test data set (*e.g.*, *test-plan-aws-pv* for the AWS PV instance types) and the benchmark cookbook (*e.g.*, *rmit-combined*). It also specifies benchmark attributes such as the load generator used for this benchmark definition. The execution schedule in the right sidebar expresses in Cron syntax at what times a new execution is triggered. The example schedule in Figure 4.6 triggers a new execution 8 times a day (*i.e.*, at 1am, 4am, 7am, 10am, 1pm, 4pm, 7pm, and 10pm).

Figure 4.7 illustrates the interactions when the RMIT benchmark is triggered by the scheduler. Following the execution design of CWB [SLCG14, SCLG15], the CWB server acquires the cloud VM including its subsidiary resources (*e.g.*, storage, private and public IPs). As soon as the cloud VM is reachable via SSH, the CWB server initiates the provisioning (*i.e.*, installation and configuration of all benchmarks) of the cloud VM via the Chef Client[52]. This client agent obtains the latest benchmark configuration from the provisioning service (*e.g.*, Chef Server[53]). The obtained configuration is applied to the cloud VM to prepare the VM for subsequent benchmark execution. The CWB server asynchronously starts the RMIT benchmark suite (4.2.1), which controls the execution of all micro and application benchmarks. Finally, the cloud VM notifies the CWB server upon benchmark completion such that the CWB server can release the VM resources via the provider API.

The resulting collection of performance metrics is stored in the CWB server and the detailed WPBench log files remain on the load generator. The majority of metrics is submitted to the CWB server during benchmark execution and stored in a relational database. They can be exported as a Comma-Separated Values (CSV) file or inspected via the CWB Web interface. For more in-depth analyses, the WPBench log files contains entries for every single HTTP request and system-level resource traces (4.2.4) at 1 second resolution.

[51] `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html`
[52] `https://docs.chef.io/chef_client.html`
[53] `https://docs.chef.io/server_components.html`

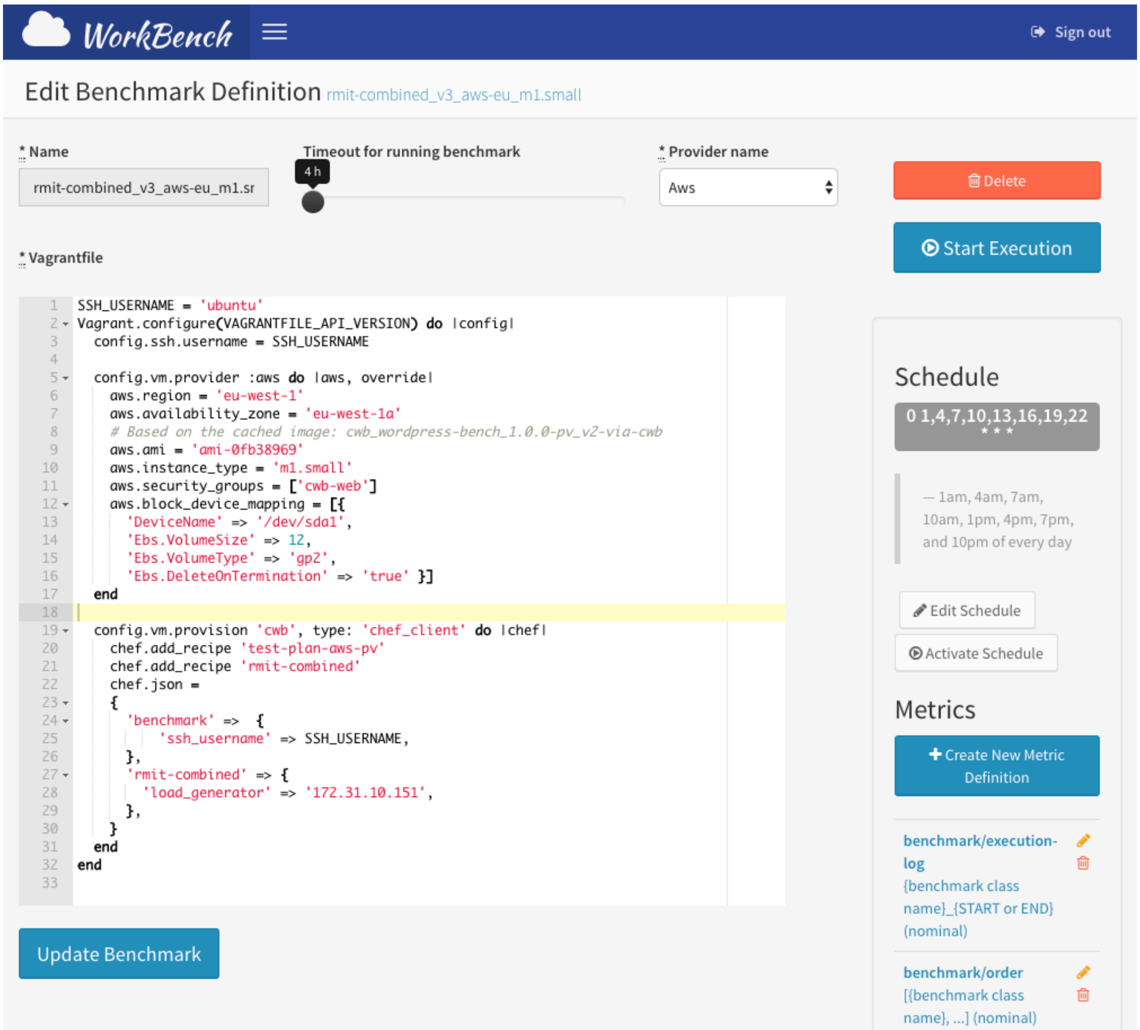

Figure 4.6: CWB Benchmark Definition

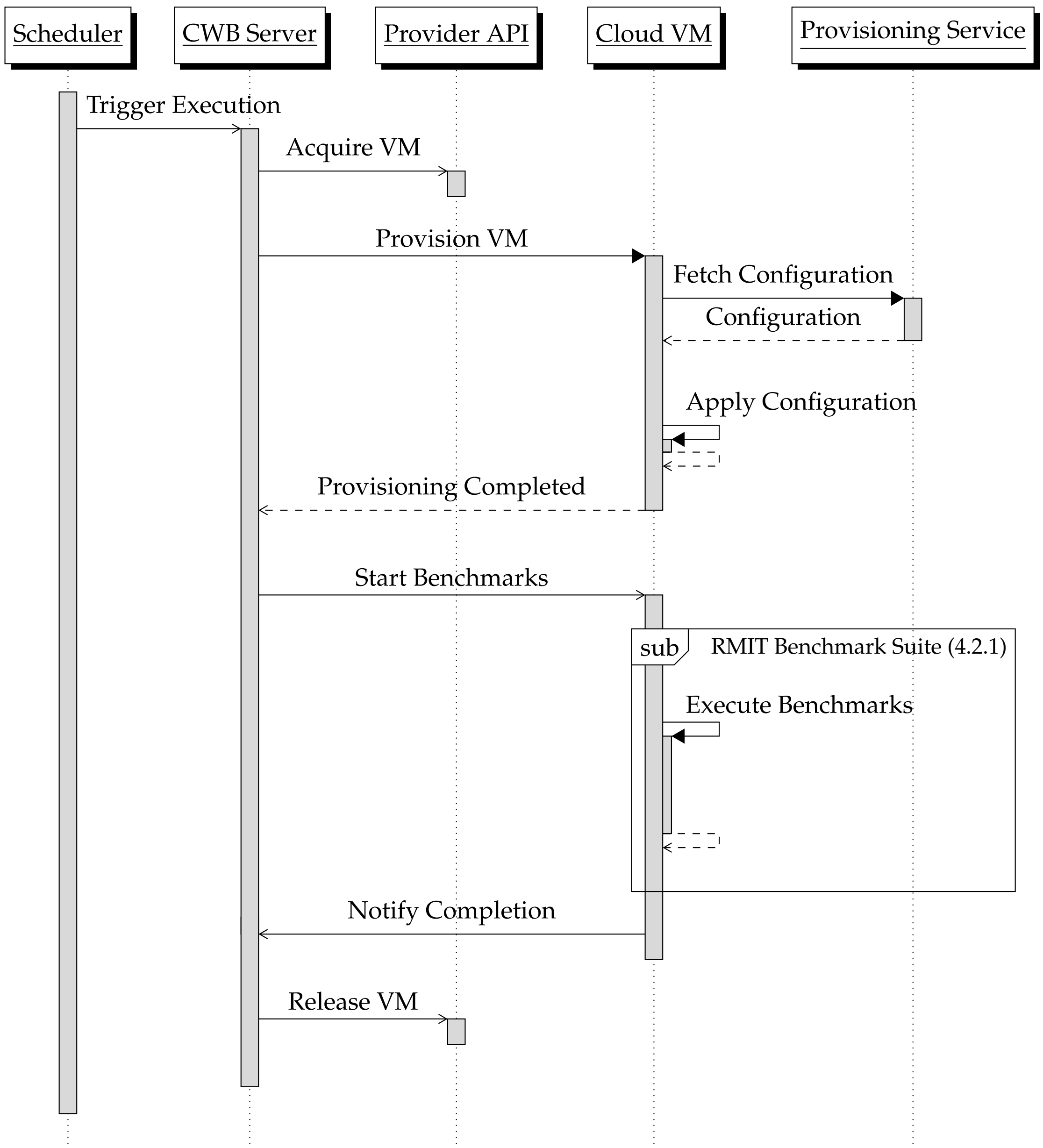

Figure 4.7: RMIT Combined Execution

4.4 Data Pre-Processing

This section describes how the performance data is pre-processed for the main analyses, which are detailed in the results Chapter 5. After briefly summarizing the export of the raw data, the pre-processing steps filtering, pivoting, and cleaning are described. The data science Integrated Development Environment (IDE) RapidMiner Studio[54] is used to model and implement this pre-processing pipeline. A supplementary shell script automates the entire pipeline by leveraging dynamically configurable macros. All scripts as well as the input and output data set are documented and freely available on Github[55].

To obtain the raw metrics data from the CWB server in an appropriate CSV format, an enhanced CSV exporter script was written in Ruby to combine and export all metrics from multiple CWB benchmarks (*i.e.*, instance types) into a single CSV file. This exporter script also assigns benchmark iterations on the same instance (*e.g.*, 1-3) for every metric result entry based on the timestamp order.

Figure 4.8 visualizes the top-level process for data pre-processing. This process reads the previously exported raw metrics and produces an interim data set. Firstly, the 1 *Filter* sub-process (Figure 4.9) removes irrelevant data for the main analyses. It discards failed or other non-finished CWB executions, iteration numbers above 3 originating from detailed execution logging, and columns with static (*e.g.*, manually defined unit from CWB) or redundant information (*e.g.*, execution status because previous filtering only included finished executions).

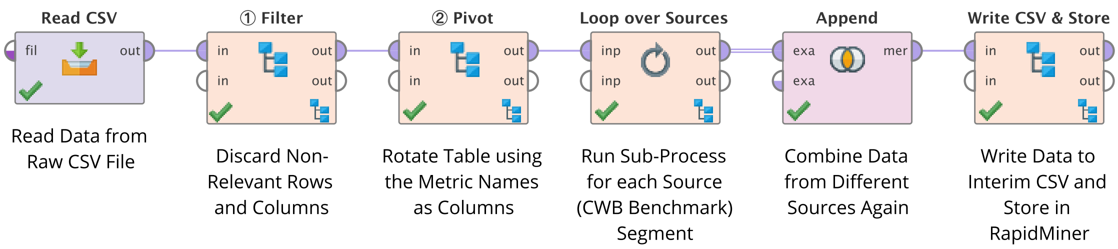

Figure 4.8: Top-Level Process for Data Pre-Processing

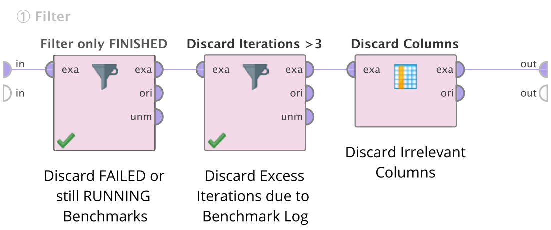

Figure 4.9: 1 Filtering Sub-Process

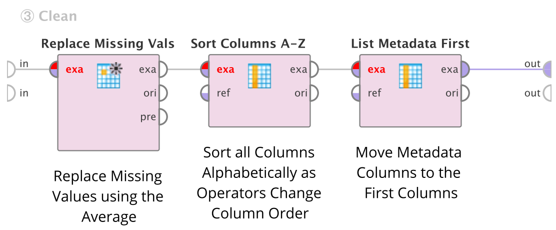

Figure 4.10: 3 Cleaning Sub-Process

Secondly, the 2 *Pivot* sub-process (Figure 4.11) rotates the tabular data such that the unique values of the metrics column are converted to new columns. A new row identifier column is

[54] `https://rapidminer.com/products/studio/`
[55] `https://github.com/joe4dev/cwb-analysis`

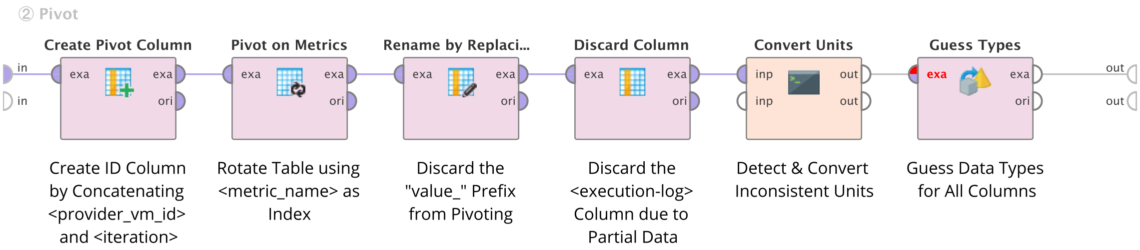

Figure 4.11: ② Pivoting Sub-Process

created by concatenating the VM identifier of the provider and the benchmark iteration counter. The pivot operator uses this new *provider_vm_id_iteration* column as the group attribute and the *metric_name* column as index attribute as illustrated in the pivot schema shown by Figure 4.12. To keep the new column names consistent with the values from the metrics column, the *value_* column name prefix introduced by the pivot operator is removed. Additionally, the remainders of the unused execution log is removed by discarding this irrelevant column. Subsequently, the custom *Convert Units* operator coverts inconsistent units or throws an exception upon detecting unresolvable inconsistencies. For a subset of metric columns (*e.g.*, excluding metadata or version columns), this operator checks whether all values are expressed in the same unit. While values with consistent units are kept, the operator attempts to apply a set of known conversions (*e.g.*, "Kb/sec" to "Mb/sec") for values with inconsistent units. It yields the converted value for successful conversions or throws an exception otherwise. Thus, the operator ensures unit consistency and can safely isolate the values by discarding the unit string. This operator is implemented as a Groovy script using regular expression pattern matching. The last step of the pivot sub-process guesses the data types (*e.g.*, integer or polynomial) for the new tabular schema obtained through pivoting.

Thirdly, the *Loop over Source* sub-process segments the entire metric collection into N groups according to their CWB benchmark names (i.e., sources or instance types) and executes the ③ *Clean* sub-process (Figure 4.10) for each of these groups. This is necessary because some operations only have meaningful semantics if they are applied on a per benchmark basis (*i.e.*, per instance type). Such a constraint is exemplified by the replacement of missing values in the first step of the cleaning sub-process. The intra-instance type variability is presumably small enough to use an average value for replacing few missing values. However, this operation wouldn't yield meaningful values if performed over the entire data set across different instance types. The subsequent steps reorder the columns alphabetically and move the special non-metric columns (*e.g.*, the identifier column *provider_vm_id_iteration*) to the front. The output of all cleaning sub-processes is combined again via the *Append* operator. Finally, the pre-processed data is written to an interim CSV file and stored in an enriched format within the local RapidMiner data repository.

4.5 Threats to Validity

This section discusses the threats to validity along the following common categories in empirical research: construct validity, internal validity, and external validity [KPP+02, WKP10, Yin08]. Additionally, reproducibility is addressed because of its particular importance in the field of cloud benchmarking.

provider_vm_id_iteration	source	metric_name	value
i-0dc663e8348856250_1.0	rmit_m1.small	sysbench/version	sysbench 0.4.12
i-0dc663e8348856250_1.0	rmit_m1.small	sysbench/cpu	95.7542s
i-0dc663e8348856250_2.0	rmit_m1.small	sysbench/cpu	95.8158s
i-0dc663e8348856250_3.0	rmit_m1.small	sysbench/cpu	95.7745s
i-0dc663e8348856250_1.0	rmit_m1.small	sysbench/fileio	1.1661 Mb/sec
i-0dc663e8348856250_2.0	rmit_m1.small	sysbench/fileio	949.19 Kb/sec
i-0dc663e8348856250_3.0	rmit_m1.small	sysbench/fileio	1.2467 Mb/sec
...	...	...	...

Group Column Metadata Column Index Column

provider_vm_id_iteration	source	sysbench/version	sysbench/cpu	sysbench/fileio	...
i-0dc663e8348856250_1.0	rmit_m1.small	sysbench 0.4.12	95.7542s	1.1661 Mb/sec	...
i-0dc663e8348856250_2.0	rmit_m1.small	?	95.8158s	949.19 Kb/sec	...
i-0dc663e8348856250_3.0	rmit_m1.small	?	95.7745s	1.2467 Mb/sec	...
...	...	...	...	...	

Figure 4.12: Pivot Schema

4.5.1 Construct Validity

Construct validity refers to the extent to which the methodology actually measures parameters relevant to the research questions.

In the context of RQ1 and given the instance type is a controlled variable, the independent variable is the individual VM instance acquired from the cloud provider (identifiable via the *provider_vm_id*) and the dependent variable is the measured performance level on a particular instance. Therefore, construct validity is the extent to which the micro and application benchmarks represent the actual VM performance. As an example, a benchmark that yields a random number would result in particularly low construct validity, whereas a benchmark that entirely saturates the CPU of an instance and correctly measures this peak performance would result in high construct validity. To mitigate this threat, benchmark-specific guidelines are followed for their configuration and the rationals behind the parameters are explained in the methodology Chapter 4. Furthermore, general performance benchmarking methodologies, such as active benchmarking [Gre13] (*cf.*, Section 4.2.4), are implemented. Several benchmarks report their resource utilization and provide additional confidence that the benchmark actually stresses the SUT. As an example, the FIO I/O benchmark reports disk utilization rates beyond 97% for most instance types, except for a few old instance types, which still achieve utilization rates above 88% for the read and above 98% for the write scenario.

In the context of RQ2, the independent variable is the instance type and the dependent variable is the measured performance level on a particular instance. Therefore, construct validity is the extent to which the benchmarks capture varying performance between different instance types. To mitigate this threat, a large set of benchmarks covers multiple resource domains (*i.e.*, computation, I/O, network, RAM) and several different aspects within each domain (*cf.*, Section 4.2.2).

One of the biggest threats with benchmarks is that they test or measure something different than intended. Anecdotally, an expert in the field provocatively claimed that almost 100% of the benchmarking reports are actually wrong because benchmarking is "very very error-prone"[56].

[56] https://www.youtube.com/watch?v=vm1GJMp0QN4&feature=youtu.be&t=18m29s

This threat does not affect the estimation model because benchmarks are treated as black box. However, it may lead to false conclusions in root cause analysis such as erroneously identifying CPU performance as the bottleneck due to a designated CPU benchmark, which is actually memory-bound[57]. To mitigate this threat, the benchmarks are carefully designed according to guidelines from research and industry, their parameters are rationalized in the methodology Chapter 4, and their implementations are publicly available for inspection on Github[58].

4.5.2 Internal Validity

Internal validity refers to the extent to which changes of the dependent variable may have been attributed to the existence of confounding variables instead of the modeled independent variable.

In the context of this study, internal validity is the extent to which cloud environmental factors, such as multi-tenancy, evolving infrastructure, or dynamic resource limits, affect the performance level of a VM instance. This is typically the biggest threat in cloud benchmarking studies because such confounding factors are oftentimes not only out of control for the experimenter but also not even measurable or known. Therefore, RQ1 is dedicated to investigate the cumulative effect of ubiquitous confounding factors on benchmark performance in terms of intra-instance-type variability. However, although these results can serve as a temporary approximation, this short-term study could have still been subjected to longitudinal patterns (*e.g.*, monthly or yearly load peaks in EC2), whose investigation were out of scope of this thesis and left for future work. Periodic patterns regarding intra-instance iterations are addressed by implementing the RMIT execution methodology [AB17] as described in Section 4.2.1. Furthermore, to mitigate interferences during benchmark execution in the VM under test, other processes (*e.g.*, cron) are terminated and periodic tasks (*e.g.*, apt package updater) are disabled. Nevertheless, the resource monitoring overhead might still cause certain interference during WPBench execution.

4.5.3 External Validity

External validity refers to the extent to which the results are generalizable to observations throughout the study domain beyond those under immediate observation.

The three most relevant threats to external validity are to what extent the results are generalizable to other cloud providers, larger instance types, and other application domains. Being the market leader for many years [Dor16], EC2 was the obvious choice as a cloud provider. Furthermore, its most extensive offer in terms of different instances types and comparability to a large body of prior work makes EC2 best suitable for this study. However, the results need to be validated for other providers, which is viable with manageable time effort because of the high automation-level of the presented methodology and design for the provider-agnostic CWB tooling. While this study almost fully covers instance types ranging from low-tier to high-tier, the extra large instance types were not considered in the study to keep experimentation costs at a reasonable level.

This study is limited to two applications from distinct domains and further experimentation is required to investigate whether suitable micro benchmarks can be found for applications in other domains. Additional Web serving benchmarks are demanded to investigate to what extent this large and heterogeneous domain is comparable with the newly crafted WPBench. The results for MDSim, serving as a representative for scientific computing applications, are speculatively widely applicable within this domain because of its similar nature to micro benchmarks. Overall, more applications have to cover other domains such as data analytics, data serving, Web

[57] `http://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html`
[58] `https://github.com/sealuzh/cwb-benchmarks`

search, or media streaming. Although the large set of micro benchmarks already broadly covers many system resources, additional multi-thread scenarios could improve the generalizability across more instance types.

4.5.4 Reproducibility

Reproducibility, sometimes called reliability [Yin08], refers to the extent to which the methodology and analysis is repeatable at any time for anyone and thereby leads to the same conclusions. Reproducibility is of utmost importance in cloud benchmarking because of the inherent dynamicity of the cloud environments themselves. Changes in the cloud environment can make it impossible to obtain the same results at another time. Therefore, cloud benchmarking methodologies need to be designed, implemented, and tested carefully to eliminate any methodological errors. Thus, whenever different results are observed applying the same methodology, these differences can be attributed to changes in the cloud environment itself and are not caused by any methodological errors. To mitigate this threat, the methodology is highly automated and together with the performance data set publicly available. Repeated benchmark executions are fully automated via CWB and thus avoid any human execution error. Merely a few initial one-time setup steps are required, such as running the WPBench test plan migration script. All tooling and benchmarks to repeat this study are publicly available as open source software[59]. Furthermore, the entire analysis is publicly available on Github[60] including documented raw and interim data sets as well as analysis and automation scripts.

[59] https://github.com/sealuzh/cwb-benchmarks
[60] https://github.com/joe4dev/cwb-analysis/

Chapter 5

Results

This chapter introduces the benchmarking data set and presents, discusses, and summarizes the results guided by the research questions introduced in Chapter 1.

5.1 Benchmarking Data Set

Using the methodology introduced in the previous chapter, a benchmarking data set was collected for the Amazon EC2 cloud provider. All configurations build upon the officially maintained Ubuntu 14.04 LTS images[1]. The exact releases[2] depend on the virtualization technology and are *ami-acb59bdf* (eu-west-1) as of April 1, 2017 for HVM instances and *ami-dd26a5cb* (us-east-1) as of April 4, 2017 for PV instances. Furthermore, the general purpose storage type *gp2* is attached to every instance because AWS recommends this type for most workloads[3].

5.1.1 Instance Type Specifications

Table 5.1 lists the specifications for the EC2 instance types in this study. It includes all available (as of April 2017) non-bursting instance types with a memory size below 15 GiB, except for c1.medium which consistently failed during experimentation for an unknown reason. This RAM threshold was chosen to keep experimentation cost at a reasonable level because the I/O workload grows substantially with increasing RAM size. The mixture between PV-based legacy instance types and more modern HVM-based instance types allows for fair comparison with prior research and adequate consideration of contemporary technology. Table 5.1 also provides the EC2 Compute Unit (ECU) specification, which Amazon used to promote as their own relative measure for CPU performance. An ECU is equivalent to the CPU power of a *m1.small* instance or a 1.0-1.2 GHz 2007 Opteron or Xeon processor type [OIY+10]. Amazon claims to conduct benchmarking to align the ECU measure with CPU power in particular regarding integer operations[4]. However, AWS quietly discontinued this approach in 2014 and moved to a more traditional way, as customary in on-premise data centers, of specifying the number of vCPUs and the type of processor[5]. The ECU model is insufficient to describe the family of general purpose instance types that follow a formal model for burstable CPU performance [LS15]. These bursting instance types

[1] `https://cloud-images.ubuntu.com/locator/ec2/`

[2] `https://cloud-images.ubuntu.com/query/trusty/server/released.txt`

[3] `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html`

[4] `https://aws.amazon.com/ec2/faqs/#What_is_an_EC2_Compute_Unit_and_why_did_you_introduce_it`

[5] `http://blogs.gartner.com/kyle-hilgendorf/2014/04/16/aws-moves-from-ecu-to-vcpu/`

Instance Type	vCPU	ECU	RAM [GiB]	Virtualization	Network Performance
m1.small	1	1	1.7	PV	Low
m1.medium	1	2	3.75	PV	Moderate
m3.medium	1	3	3.75	PV /HVM	Moderate
m1.large	2	4	7.5	PV	Moderate
m3.large	2	6.5	7.5	HVM	Moderate
m4.large	2	6.5	8.0	HVM	Moderate
c3.large	2	7	3.75	HVM	Moderate
c4.large	2	8	3.75	HVM	Moderate
c3.xlarge	4	14	7.5	HVM	Moderate
c4.xlarge	4	16	7.5	HVM	High
c1.xlarge	8	20	7	PV	High

Table 5.1: EC2 Instance Type Specifications[6]

are not included in this study because their inherently varying performance impedes controlled benchmarking.

5.1.2 Configurations and Sample Sizes

Table 5.2 shows the region-dependent instance specifications and sample sizes for each configuration. Each previously introduced instance type is tested at least once in a European data center and a subset is also tested in a North American data center. The regions *eu-west-1* (Ireland) and *us-east-1* (N. Virginia) were chosen to compare the results with prior work [LC16]. Correspondingly, the AZ "*a*" is used consistently across all regions. Notice that the hourly costs in the European region are ~3-13% higher compared to the North American region. The tailing columns in Table 5.2 report the number of benchmark executions and the resulting number of measurements including their totals. Each configuration is scheduled to execute once every 3 hours (*i.e.*, 8 times per day) and runs 3 iterations. Every iteration takes between 45 and 70 minutes depending on the instance type. This corresponds to almost continuous execution on a rolling basis (*i.e.*, a new instance is acquired once the previous instance is released) between 4 to 8 days for two low-tier, two medium-tier, and one large-tier instance type. All measurements were collected between April and May 2017.

5.1.3 Missing Values

The missing values observed for this data set during pre-processing can be categorized into three severity levels. They can be expected by design, easily replaceable, or imputable with side effect. Firstly, some instance-specific metrics are submitted once per overall benchmark execution and therefore have missing values for the second and third iteration. These metrics comprise the instance metadata (*cf.*, Table 4.1), benchmark version numbers (*cf.*, Table 4.2), and the RMIT

[6] http://www.ec2instances.info/
https://aws.amazon.com/ec2/instance-types/
https://aws.amazon.com/ec2/previous-generation/

Instance Type	Region / AZ	Cost / h [USD][*]	Executions	Measurements
m1.small	eu-west-1a	0.047	35	9030
m1.small	us-east-1a	0.044	33	8514
m1.medium	eu-west-1a	0.095	1	258
m3.medium (pv)	eu-west-1a	0.073	1	258
m3.medium (hvm)	eu-west-1a	0.073	61	15738
m3.medium (hvm)	us-east-1a	0.067	35	9030
m1.large	eu-west-1a	0.190	1	258
m3.large	eu-west-1a	0.146	58	14964
m3.large	us-east-1a	0.133	1	258
m4.large	eu-west-1a	0.111	3	774
m4.large	us-east-1a	0.108	1	258
c3.large	eu-west-1a	0.120	1	258
c4.large	eu-west-1a	0.113	4	1032
c4.large	us-east-1a	0.100	3	774
c3.xlarge	eu-west-1a	0.239	1	258
c4.xlarge	eu-west-1a	0.226	3	774
c4.xlarge	us-east-1a	0.199	1	258
c1.xlarge	eu-west-1a	0.592	1	258
[*]Linux On-Demand as of 2017-05-19		Total	244	62952

Table 5.2: Specification and Sample Sizes per Configuration

Benchmark Metric	# Missing # Total	RSD min [%] RSD max [%]	Instance Type Region / AZ
fio/4k-seq-write Latency	1 105	10	m3.medium us-east-1a
sysbench/fileio-4k-rand-write Throughput	5 99	14	m1.small us-east-1a
fio/4k-seq-write Duration	348 684	5 10	multiple
sysbench/fileio-4k-rand-read-prepare Duration	348 684	1 4	multiple

Table 5.3: Missing Values

benchmark execution order (*cf.*, Section 4.2.1). Secondly, some metrics report static values by design but were introduced after the start of the experiment to consistently report a duration value for every benchmark. The missing values of these three metrics can be replaced easily with their well-known execution times from the benchmark design using the constants *60000* for the duration of the fio/8k-rand-read benchmark (*cf.*, 4.2.2) and with *30* for the durations of the single- and multi-thread iperf benchmarks (*cf.*, 4.2.2). Thirdly, a few metrics report values that are inherently dynamic and cannot be replaced without selectively affecting the nature of the data distribution.

Therefore, Table 5.3 reports these dynamic metrics with missing values in more detail. The single missing value for the latency of the *fio/4k-seq-write* benchmark might be caused by erroneous benchmark output or unsuccessful metric submission. However, replacing this single value with the average out of the 104 remaining samples should not affect the data distribution much as it constitutes less than 1% of the samples. The 5 missing throughput values for the *sysbench/fileio-4k-rand-write* were caused due to value-dependent unit reporting of the Sysbench tool, which was not considered in the metric extraction at first. Sysbench switches its default reporting unit from *Mb/sec* to *Kb/sec* for values below 1 Mb/sec. Before this adjustment to the metric extraction was applied, the values below 1 Mb/sec failed to match the regular expression and were thus ignored causing these missing values. Using the average replacement method would skew this data towards a higher average and lower Standard Deviation (SD). Therefore, the average from all samples below 1 Mb/sec is used as a more adequate replacement value. The 348 missing values for the durations of the FIO and Sysbench I/O benchmarks originated from the fact that these metrics were introduced after the start of the experiment to consistently report a duration value for every benchmark. Their large fraction of missing values (~50% of the overall samples) and non-neglectable Relative Standard Deviation (RSD) (1-10% grouped by instance type and region configuration) definitely influence the shape of the data when using the average replacement method. If these duration values would be important for the analysis part, a more robust replacement method must be chosen (*e.g.*, predicting the duration from correlated attributes such as bandwidth).

5.2 RQ1 – Performance Variability within Instance Types

This section outlines the approach, presents and discusses the results, draws its implications, and summarizes the findings for RQ1:

RQ1 – Performance Variability *within Instance Types*

Does the performance of equally configured cloud instances vary relevantly?

5.2.1 Approach

To answer this research question, the relevant subset of data is prepared and the variability is assessed by calculating the RSDs as formally defined in Equation 5.1

$$\mathrm{RSD} = 100 \cdot \frac{\sigma_m}{\overline{m}} \tag{5.1}$$

where σ_m is the absolute standard deviation and $\overline{m}$ is the arithmetic mean of the metric m.

Starting from the extensive interim data set (*cf.*, Section 5.1 and Figure 4.1), the iterations are aggregated, the relevant metrics are selected, and the relevant samples are filtered. Iteration aggregation groups all samples by the unique provider instance id, which every VM instance obtains when being acquired. Calculating the average for numerical metrics or the mode (*i.e.*, most often appearing value) for nominal metrics allows to compare the performance of different instances of the same instance type. Metric selection reduces the originally 86 metrics to 38 relevant metrics to answer this research question. Most of the ignored metrics are static by design for the RMIT benchmark (*e.g.*, version numbers, fixed workload durations) and for the instance type (*e.g.*, number of CPU cores, available memory). Others include execution metadata (*e.g.*, RMIT benchmark order), runtime statistics (*e.g.*, disk utilization during FIO benchmark), non-mean values (*e.g.*, 95% percentiles), or redundant metrics (*e.g.*, IOPS because of bandwidth). The redundant metrics are identified by calculating all pairwise correlations. Table 5.4 lists the selected and discarded metrics exhibiting perfect correlation (*i.e.*, $\rho = 1$) according to the Pearson Correlation Coefficient (PCC). Sample filtering only considers configurations with more than 30 samples as *relevant* for this analysis and ensures sample size consistency. These filtered configurations focus on smaller instance types because prior work has shown that they tend to deliver less stable performance than larger instance types [LC16, WN10, Kot14] besides being more cost-efficient to benchmark. Furthermore, randomized sub-sampling is applied to ensure that every configuration uses the same number of samples (*i.e.*, 33). The use of a local random seed ensures reproducibility of the sampling process. All these steps are implemented as RapidMiner processes and automated via a shell script[7].

To assess overall performance variability, the distribution of the RSDs is summarized for each relevant configuration using a combined violin and dot plot to attribute for its non-normal distribution. An RSD is considered to be *relevant* if it exceeds the threshold of 5%, following the definition of a large benchmarking study [LC16]. These steps are implemented as an RScript[8] and also available on Github[9].

[7] `https://github.com/joe4dev/cwb-analysis/tree/master/rq1`
[8] `https://www.r-project.org/`
[9] `https://github.com/joe4dev/cwb-analysis/blob/master/rq1/rsd-plots.R`

Selected	Discarded
fio/4k-seq-write-bandwidth	fio/4k-seq-write-iops
fio/8k-rand-read-iops	fio/8k-rand-read-bandwidth
sysbench/mutex-latency	sysbench/mutex-duration
sysbench/threads-1-latency	sysbench/threads-1-duration
sysbench/threads-128-latency	sysbench/threads-128-duration

Table 5.4: Redundant Metrics

5.2.2 Results

Figure 5.1 summarizes the variability in terms of RSD for each relevant configuration using violin plots with annotated mean values. All medians are clearly below the 5% threshold and almost all means, denoted by the blue diamond, lie underneath this relevant variability threshold. Only the mean for the configuration *m3.large (eu)* exceeds the 5% threshold due to few clear outliers that exhibit large distances (factor 8-20) from the median. Thus, performance does not vary relevantly for the majority of benchmarks in all these tested configurations.

5.2.3 Discussion

This result is fairly surprising and contrasts the findings of prior work. Many benchmarking studies repeatedly confirmed large variability in performance between supposedly identical instances [FJV$^+$12, OZL$^+$13, SDQR10, LYKZ10, DPC10, CGPS12, WN10, BS10, EKKJP10, OZN$^+$12]. Concerning Amazon EC2, all these studies exclusively focus on instance types of the first generation[10]. A more recent study [LC16] additionally included three second generation instance types and has shown that their performance is considerably more stable according to their experiments conducted between July and August in 2014. Taking *m3.large* as an example for such a second generation instance type, a direct comparison of the exact same CPU benchmark (*i.e.*, Sysbench Single Thread) revealed that their RSD is identical at a very predictable level of 0.13%. For the first generation instance type *m1.small* (Amazon's oldest instance type announced in 2006[10]), the same direct comparison indicates more stable CPU performance due to eliminated hardware heterogeneity. Consistently serving the same CPU models could reduce the RSD from 3.19% (2014) to 0.25% (2017) in the European region and from 12.81% to 0.30% in the North American region. Nevertheless, its more than two times higher RSD compared to larger instance types such as m3.large is presumably caused by noisy neighbors due to resource sharing of the underlying hardware for small instance types such as m1.small. Amazon confirms the presence of shared resources for m1.small when prohibiting vulnerability and penetration testing on this instance type[11].

File I/O performance became substantially more stable moving from HDD-backed storage to SSD-backed storage. Although using the same benchmark tool as in [LC16], the File I/O results are not directly comparable because they tested a combined read and write workload on HDD storage while this study tests different I/O types (write/read), I/O modes (sequential/random), and block sizes (1m/4k) on SSD storage. Nevertheless, contrasting to their observed substantial

[10] `https://aws.amazon.com/blogs/aws/ec2-instance-history/`
`https://aws.amazon.com/blogs/aws/new-ec2-second-generation-standard-instances-and-price-reductions-1/`

[11] `https://aws.amazon.com/security/penetration-testing/`

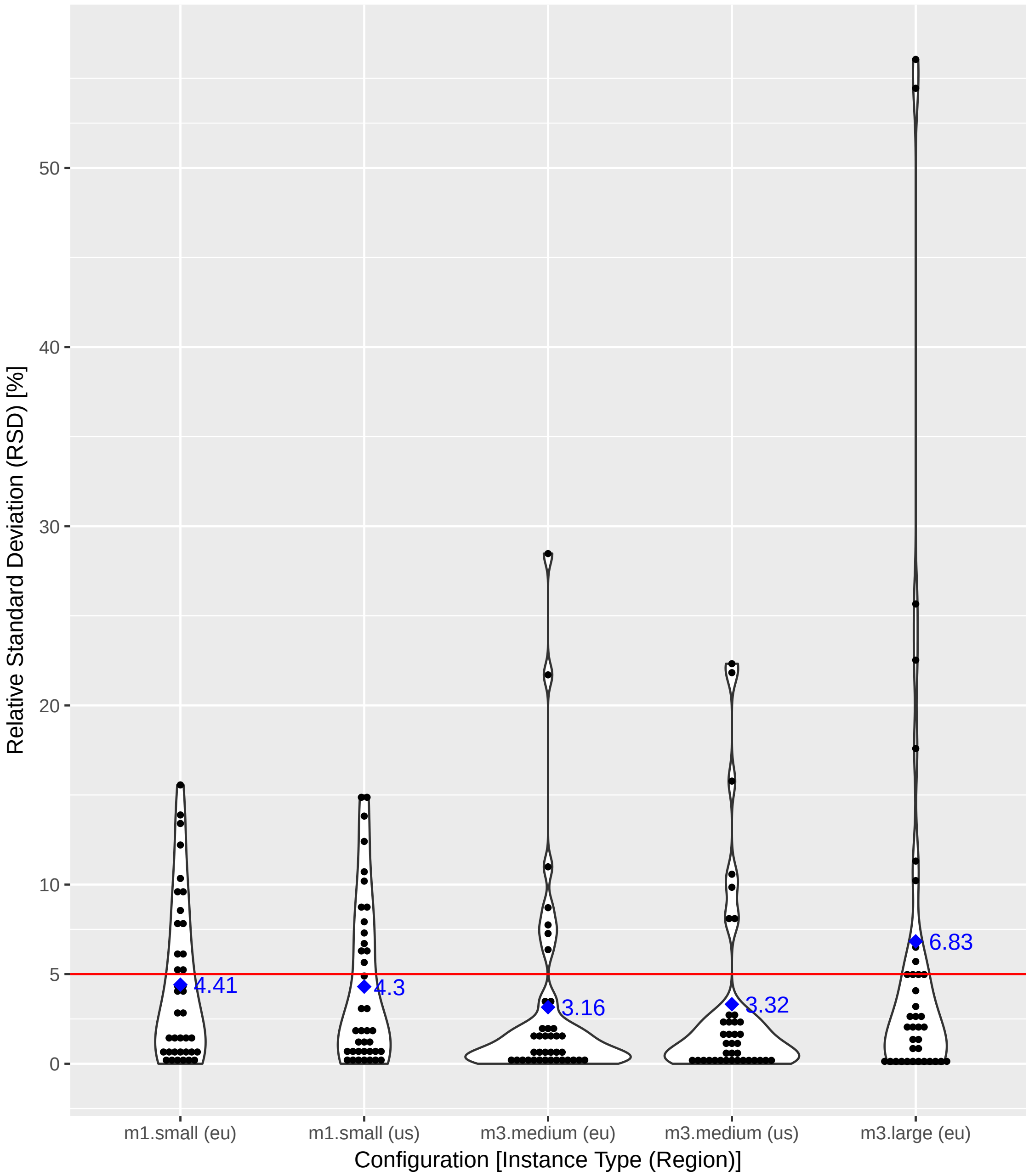

Figure 5.1: Variability per Configuration

variability ranging from 20% to almost 100% RSD, sequential write throughput with larger block size performed remarkably stable with RSDs mostly below 1.5%. Scenarios with smaller block sizes and random I/O mode inherently performed less stable with RSDs in the intervals $[5, 14]\%$ for write and $[12, 22]\%$ for read I/O types. The lower level I/O benchmark FIO confirmed this generally low variability with RSDs below 10% for both of its I/O scenarios: 4k Sequential Write and 8k Random Read.

Most surprisingly, network performance achieved almost perfect stability, which contrasts the 25% RSD observed several years ago between March and April in 2012 [FJV$^+$12]. Presumably, AWS fundamentally changed their approach to intra-AZ networking and might perform customer-based placement optimizations using strategies such as placement groups[12].

Amazon's shifts towards delivering more stable performance has been also observed in another recent experiment with a Web serving workload [DISL17], which yielded RSDs below the 5% threshold. The results in this thesis provide further evidence for this observation and expand its validity to a broad range of micro and application benchmarks.

5.2.4 Implications

The nowadays largely stable performance (*i.e.*, low variability) for equally configured cloud instances has several implications for researchers and practitioners.

Research presented several approaches that exploit performance variability, especially caused by hardware heterogeneity, to reduce costs up to 30% [OZL$^+$13, OZN$^+$12] or improve performance up to 5% for CPU and 35% for network workloads [FJV$^+$12]. The results of this thesis suggest that such instance seeking approaches, also called placement gaming, are not worthwhile anymore. Furthermore, cloud benchmarking studies spent a lot of resources into obtaining relevant sample sizes to achieve statistically plausible results within typical confidence intervals (*i.e.*, 95% or 99%). While a single sample is sufficient for many benchmarks to achieve the 99% confidence interval, around 10 to 20 samples are required for less stable benchmarks for the 95% confidence interval. Thus, the data indicates that benchmarking efforts can be reduced considerably because fewer sample sizes suffice especially for highly stable categories such as CPU or intra-AZ network performance. This motivates new areas of research because collecting a relevant amount of performance data for a broad range of instance types becomes more viable. For example, RQ2 investigates whether micro benchmark measurements from different configurations can profile and estimate application performance across different instance types.

For practitioners, stable performance delivers a fair offer and is attractive for variability-sensitive use cases such as running software performance test suites. In a fair offer, every cloud customer consistently obtains the same performance for equally specified services. The results indicate that customers can trust Amazon's instance type specification and do not have to be concerned about getting poorly performing instances. It also alleviates the threat that few optimizing customers (*e.g.*, Netflix allegedly[13]) obtain better performing instances than regular users. Further, software performance test suites are susceptible to platform-induced performance variability because their goal is to detect changes in performance at the code-level. Thus, lower variability reduces the interference factor and requires less iterations to detect code-level regressions with high confidence. Therefore, cloud computing with its seemingly unlimited computing resources, provides an attractive model to offload and parallelize long running software performance test suites.

[12] `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html`

[13] `https://www.reddit.com/r/aws/comments/547xbx/netflix_found_5x_performance_variation_between/`

5.2.5 Summary

The data supports that performance for equally configured cloud instances does *not* vary relevantly for most benchmarks in Amazon's EC2 cloud, neither for small, medium, and large instance types tested in two different regions. Whereas some prior work becomes inapplicable, this also opens up new avenues for future research.

5.3 RQ2 – Application Performance Estimation across Instance Types

RQ2 – Application Performance Estimation *across Instance Types*

Can a set of micro benchmarks estimate application performance for cloud instances of different configurations?

This research question addresses the feasibility of estimating application performance from micro benchmarks and is divided into two sub-questions, which are dedicated to evaluate the accuracy of the estimates (RQ2.1) and identify the most suitable micro benchmark estimators (RQ2.2). For both sub-questions, the approach, discussion, implications, and summary is presented in the following.

5.3.1 RQ2.1 – Estimation Accuracy

RQ2.1 – Estimation Accuracy

How accurate can a set of micro benchmarks estimate application performance?

Approach

To answer this research question, the relevant subset of data is prepared and a linear regression model is trained and evaluated for every application benchmark.

Starting from the extensive interim data set, data preparation selects the relevant metrics, enhances the data set with instance type metadata, filters the relevant samples, and labels training and test data. Metric selection follows the same procedure as described for RQ1 (5.2.1). Data enrichment then maps the CWB benchmark name to instance type metadata such as its API name (*e.g.*, *m1.small*), number of virtual CPUs, or ECU. Sample filtering selects three iterations from the same execution for each instance type in the European data center. These limited sample sizes are motivated by the findings from RQ1 and should exemplify the practical applicability of this approach. Finally, the boundary instance types (*i.e.*, the smallest and largest) are labeled as training data to capture the largest possible instance type diversity.

A forward feature selection algorithm is combined with linear regression to automatically identify the best performing set of features (*i.e.*, metrics) regarding the relative error performance criterion. Forward feature selection starts with an empty set of features and iteratively adds a previously unused feature. In a sub-process, the candidate feature set is then used to build a linear regression model with the training data. This model is applied to the test data and the mean relative error is calculated between the predicted and actual application performance. In doing so, only features that yield the highest gain for the performance criterion (*i.e.*, minimize the relative error) are kept. This sub-process is repeated until no additional feature can further improve the relative error. At the end, forward feature selection outputs a weighted feature list and various performance indicators such as the relative error or the Pearson correlation coefficient, also known as squared correlation or R^2. All these steps are implemented as RapidMiner processes and available on Github[14]. Furthermore, every prediction model was reproduced in an RScript and manually reviewed regarding its prediction outcome and visual fitting.

Results

Table 5.5 reports the estimation accuracy in terms of the relative estimation error achieved by the best micro benchmark predictor for WPBench and MDSim. For WPBench, all three scenarios (*i.e.*, read, search, write) are evaluated regarding their metrics Response Time (RT) and Throughput (TP). The boundary instance types are labeled as training data (*i.e.*, *Train*) and the relative estimation error is listed per instance type. Notice that no suitable micro benchmark could be found for the WPBench throughput metrics across all instance types. Therefore, the throughput results only include instance types with one and two virtual CPUs. Hence, these results are less significant and not directly comparable against the other application metrics. For each instance type, the averaged relative error over the three iterations indicates how far the estimated performance consistently lies above (*cf.*, +) or below (*cf.*,-) the actual performance. Wherever the actual values are spread on both sides of the regression line, the pipe (*cf.*, |) indicates the absolute error due to high variability between iterations. In summary, the mean Relative Error (RE) combined with the max RE indicates the fitness of cross instance performance estimation. The *max RE* estimates the upper bound for the relative error assuming that the smallest instance performs worst and the largest instance performs best. This provides an orientation on how far the minimum and maximum of the application performance is spread. Hence, a high max RE implies that high accuracy (*i.e.*, low relative error) is harder to achieve. Conversely, a low max RE diminishes the significance of low relative errors because they are more likely to occur by chance.

The most accurate estimates are achieved by MDSim and the read and search scenarios of WPBench as shown in Table 5.5. Duration estimates for MDSim reach 8.2% accuracy for its duration values in the interval $[69.7, 491.7]$ seconds (*cf.*, max RE of 600%). The read and search scenarios of WPBench exhibit by far the largest spread in their response time distribution in the interval $[65.8, 1457.8]$. This spread is illustrated in Figure 5.2 for the read scenario and results in a maximum relative error of 21000%. Nevertheless, moderate relative errors of 12.5% and 17.5% are achieved on average. Furthermore, these linear regression models are statistically significant at the 0.001 level and thus support the assumption of low variability shown in RQ1.

The estimation accuracies for the throughput of the WPBench read and search scenario are relatively weak given the reduced test set and therefore further limited spread in their application performance data. However, the mean relative error is strongly driven by the high overestimation (*i.e.*, lower actual throughput than estimated) of m3.medium application performance and the high underestimation (*i.e.*, higher actual throughput than estimated) of m1.large performance as illustrated in Figure 5.3. Both regression models also show statistical significance at the 0.001 level.

The relative errors for the WPBench write scenario are generally high, particularly given the relative low spread of their performance data. Additionally, even within the same instance type, application performance is overestimated and underestimated simultaneously and therefore provided as modulus value. Furthermore, their regression models are less significant at the 0.05 (response time) and the 0.1 (throughput) level, which adds further evidence for the existence

[14] `https://github.com/joe4dev/cwb-analysis/tree/master/rq2`

<table>
<tr><th></th><th colspan="6">WPBench</th><th></th></tr>
<tr><th></th><th colspan="2">Read</th><th colspan="2">Search</th><th colspan="2">Write</th><th>MDSim</th></tr>
<tr><th>Instance Type</th><th>RT</th><th>TP</th><th>RT</th><th>TP</th><th>RT</th><th>TP</th><th>Duration</th></tr>
<tr><td>m1.small</td><td>Train</td><td>Train</td><td>Train</td><td>Train</td><td>Train</td><td>Train</td><td>Train</td></tr>
<tr><td>m3.medium (pv)</td><td>+6.9</td><td>-17.9</td><td>+7.5</td><td>-21.1</td><td>|21.5|</td><td>|23.9|</td><td>+5.4</td></tr>
<tr><td>m3.medium (hvm)</td><td>+14.7</td><td>+64.3</td><td>+6.1</td><td>+64.9</td><td>|42.6|</td><td>|26.4|</td><td>+5.8</td></tr>
<tr><td>m1.medium</td><td>+9.0</td><td>-8.4</td><td>+9.0</td><td>-13.5</td><td>|36.2|</td><td>|26.1|</td><td>-0.2</td></tr>
<tr><td>m3.large</td><td>-17.8</td><td>+2.2</td><td>-25.6</td><td>+2.7</td><td>|53.1|</td><td>|33.0|</td><td>-10.2</td></tr>
<tr><td>m1.large</td><td>+17.0</td><td>-66.6</td><td>+17.5</td><td>-68.5</td><td>|40.9|</td><td>|41.6|</td><td>-0.8</td></tr>
<tr><td>c3.large</td><td>-17.4</td><td>+0.8</td><td>-26.1</td><td>+2.3</td><td>|51.5|</td><td>|32.7|</td><td>-10.1</td></tr>
<tr><td>m4.large</td><td>-3.6</td><td>+0.2</td><td>-12.8</td><td>+0.5</td><td>|52.8|</td><td>|34.1|</td><td>-12.4</td></tr>
<tr><td>c4.large</td><td>-9.7</td><td>Train</td><td>-18.4</td><td>Train</td><td>|50.3|</td><td>Train</td><td>-12.5</td></tr>
<tr><td>c3.xlarge</td><td>-26.3</td><td></td><td>-34.4</td><td></td><td>+32.8</td><td></td><td>-11.2</td></tr>
<tr><td>c4.xlarge</td><td>-2.2</td><td></td><td>-17.6</td><td></td><td>+26.1</td><td></td><td>-13.7</td></tr>
<tr><td>c1.xlarge</td><td>Train</td><td></td><td>Train</td><td></td><td>Train</td><td></td><td>Train</td></tr>
<tr><td>Mean RE</td><td>12.5</td><td>23.0</td><td>17.5</td><td>24.8</td><td>40.8</td><td>31.1</td><td>8.2</td></tr>
<tr><td>Max RE</td><td>2100</td><td>310</td><td>1810</td><td>280</td><td>140</td><td>120</td><td>600</td></tr>
</table>

Table 5.5: Relative Estimation Errors [%]

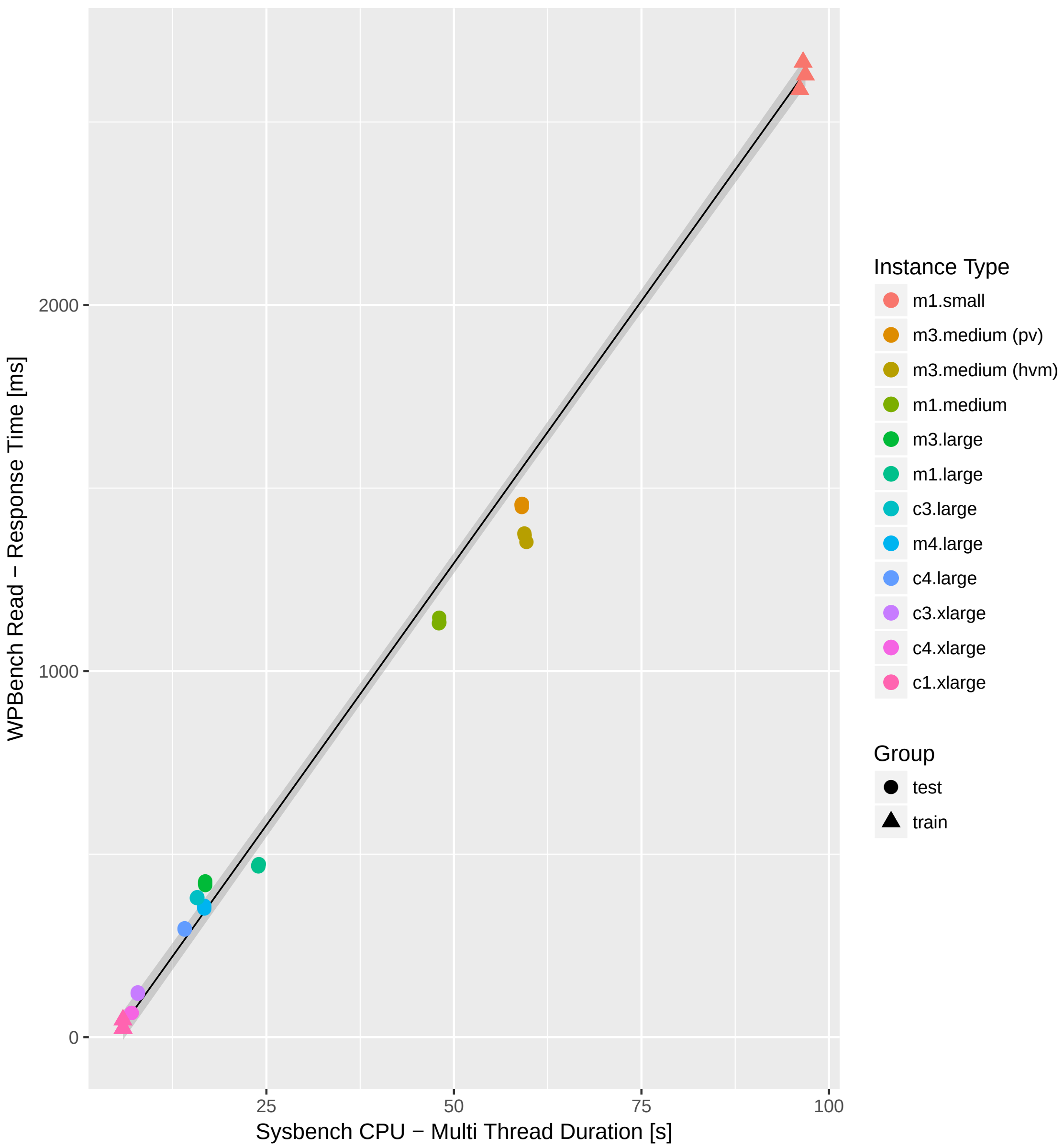

Figure 5.2: Linear Regression Model for WPBench Read – Response Time

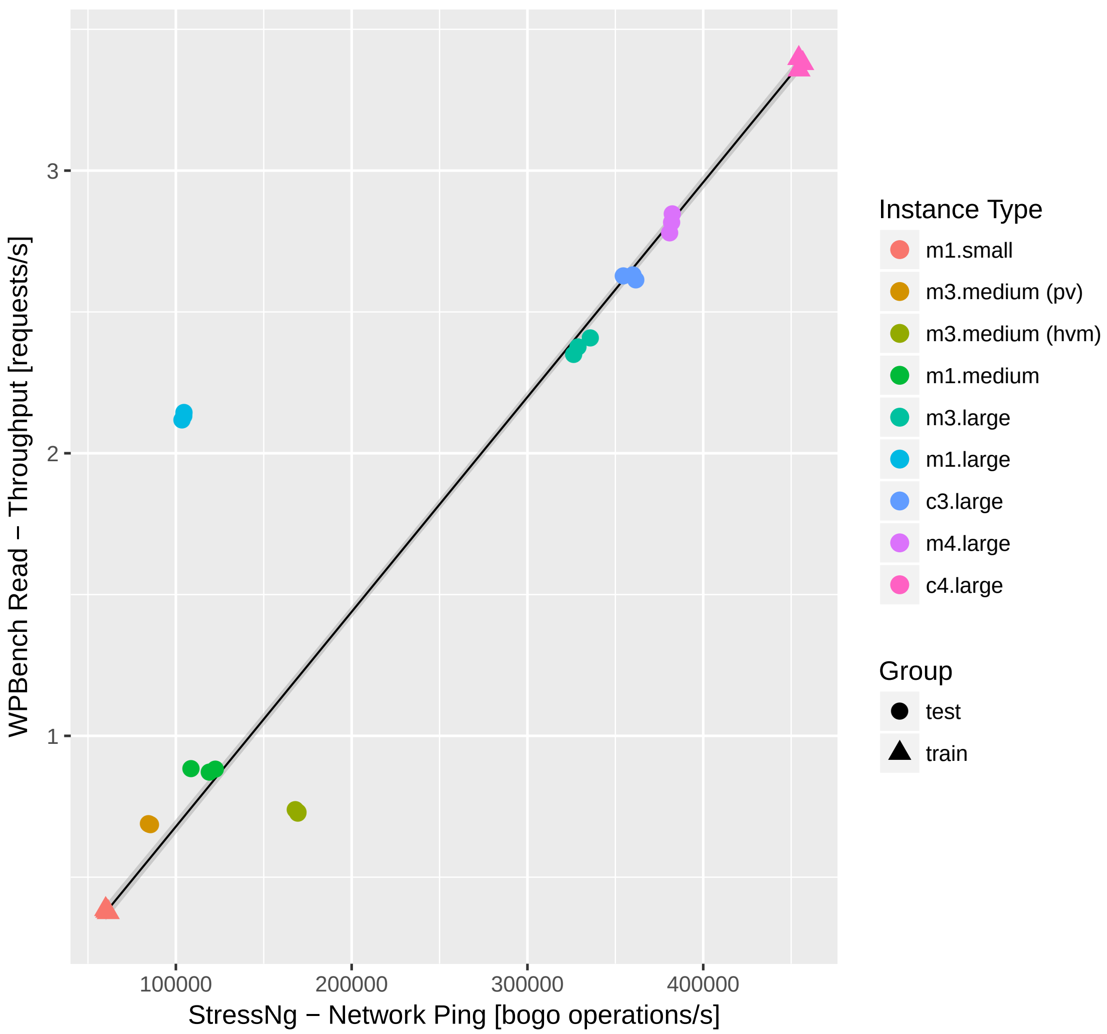

Figure 5.3: Linear Regression Model for WPBench Read – Throughput

of performance variability between different iterations. This hypothesis is further investigated by performing statistical tests for the 5 instance types used in RQ1 with relevant sample sizes between 33 and 61 executions (*cf.*, Table 5.2) from the interim data set. A One-way ANOVA test [BGT12] is performed upon the iteration column as its group attribute. The results confirm that both response time and throughput vary greatly (*i.e.*, particularly high f value) between the 3 different iterations with high significance (*i.e.*, $p < 0.001$) for all 5 tested instance types. ANOVA is an omnibus test and therefore only confirms a statistically significant difference between the iterations but does identify the specific iterations that differ statistically significant from each other. Therefore, a Mann Whitney U-Test, also called Wilcoxon rank-sum test or Wilcoxon-Mann-whitney test, is conducted to demonstrate that even the differences between all pairs of iterations are statistically significant for all 5 instance types. The increasing performance between the individual iterations becomes apparent in the linear regression model shown in Figure 5.4. Notice that the statistical tests have also shown that apart from WPBench, none of the other benchmarks exhibit statistically significant differences between iterations.

Discussion

The attribution of relative errors to instance types in Table 5.5 reveals certain patterns that might originate from differences in virtualization or processor generations between instance types. For the throughput in the WPBench read and search scenarios, the estimation for the HVM version of m3.medium overestimates application throughput by ~64%. Interestingly, its PV counterpart achieves similar application performance but performs ~50% worse in its estimator benchmark StressNg – Network Ping. This might be caused by additional latency introduced in the VMM for PV instances for privileged instructions (*i.e.*, system calls). Conversely, HVM instances can bypass the VMM for operations that are slow when being emulated such as network calls[15], which dominate the workload of the StressNg – Network Ping benchmark.

For the same WPBench scenarios, application throughput for m1.large is underestimated by ~67%. Interestingly, its official successor m3.large, following Amazon's upgrade path for previous generation instances[16], fits the regression line almost perfectly (~2.4% RE). While m3.large provides only slightly better throughput (~10%) than its predecessor, it outperforms m1.large by almost factor 3 in the network Ping benchmark. Figure 5.3 visualizes this observation with the double-vCPU instance type m1.large being on par with the single-vCPU instance types in the lower corner. Further investigation of the CPU models that are served for these instance types reveals that the turbo boost frequency of 2.8 Gigahertz (GHz)[17] for m1.large is considerably lower than for the more modern CPU model served for m3.large, which reaches 3.3 GHz[18] in turbo mode. The turbo mode can dynamically increase the clock frequency of a single CPU core as needed if other cores are idling and is thus capable of delivering higher single core performance with its additional thermal and power headroom provided by an extra CPU core[19]. This explanation conforms with the other modern two-vCPU instances types (*e.g.*, c3.large[20]) reaching even higher frequencies in turbo mode (*e.g.*, up to 3.6 GHz) by specification. Notice that the actual de-

[15] https://www.slideshare.net/AmazonWebServices/deep-dive-on-delivering-amazon-ec2-instance-performance-64919720

[16] https://aws.amazon.com/ec2/previous-generation/

[17] https://ark.intel.com/products/64590/Intel-Xeon-Processor-E5-2650-20M-Cache-2_00-GHz-8_00-GTs-Intel-QPI

[18] https://ark.intel.com/products/75275/Intel-Xeon-Processor-E5-2670-v2-25M-Cache-2_50-GHz

[19] http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

[20] http://ark.intel.com/products/75277/Intel-Xeon-Processor-E5-2680-v2-25M-Cache-2_80-GHz

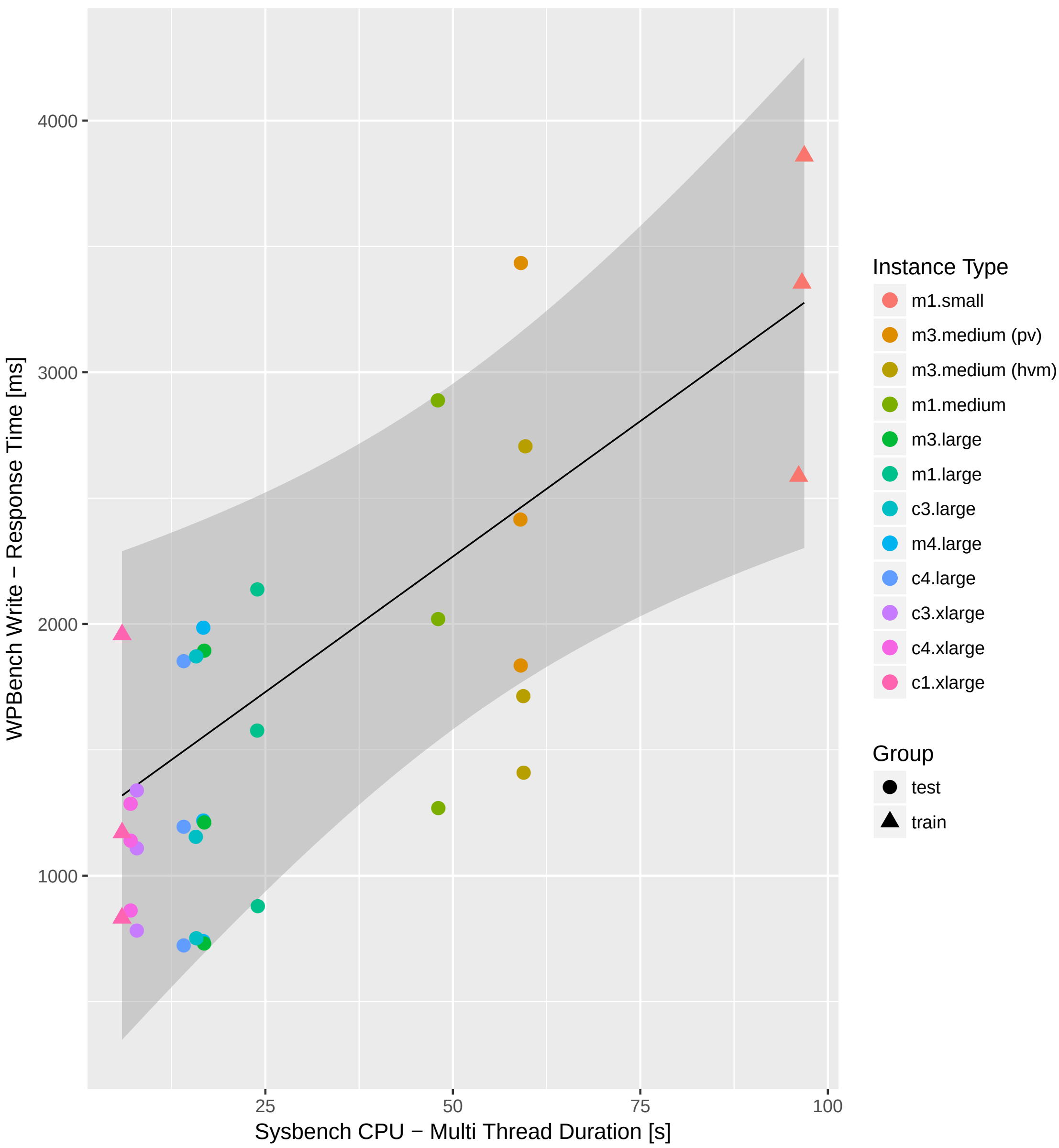

Figure 5.4: Linear Regression Model for WPBench Write – Response Time

livered CPU models might differ compared to their official manufacturer specifications because AWS operates custom designed processor models in its EC2[21] data centers.

Implications

The ability to estimate application performance with an acceptable accuracy highlights the usefulness of micro benchmarks. However, technological factors such as the type of virtualization (*i.e.*, PV vs HVM) or processor generations (low vs high turbo mode) can have a profound impact on estimation accuracy and are not captured in the linear model. It seems that modern instance types are more prone to micro benchmark-favoring optimizations and previous generation instance types are more susceptible to micro benchmark-hampering penalties. Thus, choosing a modern instance as training data could lead to general application performance underestimation, while the choice of such a previous generation instance could potentially overestimate application performance.

Benchmarks should be executed multiple times and statistical tests should be conducted to investigate whether performance varies significantly between different iterations. The write scenario of WPBench demonstrates how benchmark-induced variability between iterations severely impacts the meaningfulness of a model. Beyond benchmark-induced variability, this methodology would also detect platform-induced variability caused by bursting schemes such as the EC2 CPU bursting [22] [LS15] or EBS I/O bursting[23].

Summary

The results show that micro benchmarks are able to estimate the performance for a scientific application with a mean relative error below 10% and the response time of a Web serving application with a relative error between 10% and 20%. Throughput estimates are less accurate with a relative error around 25% and the write scenario of the Web serving benchmark suffered from benchmark-induced performance variability and thus exhibits high error rates above 30%.

5.3.2 RQ2.2 – Micro Benchmark Selection

RQ2.2 – Micro Benchmark Selection

Which subset of micro benchmarks estimates application performance most accurately?

Approach

The approach for this research question follows the feature selection process as described in the approach section for RQ2.1.

Results

For the response time across all scenarios of WPBench and the duration of MDSim, forward feature selection included the Sysbench – CPU Multi Thread micro benchmark in the linear model.

[21] https://aws.amazon.com/blogs/aws/new-c4-instances/

[22] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-instances.html#t2-instances-cpu-credits

[23] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html#EBSVolumeTypes_gp2

For the WPBench write scenario, two additional benchmarks were proposed with equal weights but rejected because their contribution to the model was statistically insignificant with p=0.393 for fio/8k-rand-read-latency and p=0.450 for fio/4k-seq-write-bandwidth. Similarly, for MDSim, the additional attribute fio/8k-rand-read-iops was discarded due to its p-value 0.0976 at the border of being insignificant. To adjust for the reduced sample size for the throughput metrics, the feature selection algorithm was modified to select the single best performing metric. Subsequently, the WPBench read and search scenario both choose the StressNg – Network ICMP Ping benchmark. As this benchmark is also in the top 5 list for the write scenario and the difference in terms of relative error is only marginal (<1.3%), it has been manually selected to ensure consistency.

Table 5.6 and Table 5.7 presents the best benchmark estimators and two instance type specification metrics serving as a baseline. For each estimator, the mean relative error with its range and the squared correlation R^2 are provided. R-squared, also known as coefficient of determination, measures how well the data fits the regression line where 0% implies that the model captures no variability in the data and 100% implies that the model perfectly fits all data on the regression line. Finally, the max RE is provided analogous to its previous definition in 5.3.1.

For the response time of the WPBench read and search scenarios and the duration of MDSim presented in Table 5.6, the multi thread Sysbench – CPU benchmark serves as a good estimator. The almost perfect fit of the regression model (*i.e.*, $R^2 > 98.9$) together with low relative errors below 10% for MDSim and between 10% and 20% for the read and write scenarios of WPBench indicate that this multi thread CPU benchmark can be a very suitable estimator. Further, the vastly inferior results for the single thread version of the same benchmark reveal that they cannot be used interchangeably. In addition, the improvements upon the vCPU and ECU baselines are substantial. Although ECU is already ~50% more accurate than using the number of vCPUs, the Sysbench benchmark outperforms this baseline by factor 17 to 29 in terms of relative error. The CPU benchmark also fits the regression line considerably better with over 33% improvement upon squared correlation compared to the baseline metrics.

For the throughput of the WPBench read and search scenarios, the StressNg – Network ICMP ping benchmark serves as moderate estimator. The ICMP benchmark works well as a WPBench throughput estimator for the majority of the instance types. However, two outliers strongly affect the mean relative error as discussed in Section 5.3.1 and illustrated in Figure 5.3. Therefore, the mean relative error is only marginally better (<5%) than the baseline and the squared correlation is even worse than the baseline metrics.

Discussion

While the results support the conjecture that these estimates could be meaningful for applications with a resource profile similar to micro benchmarks, such as MDSim, the linear model also works surprisingly well for a more diverse application such as WPBench. The MDSim application is very CPU heavy, potentially stresses the main memory, but does not involve I/O operations. Despite the fact that MDSim performs higher-level real-world computations compared to low-level artificial micro benchmark workloads, such as iterating over meaningless loops, resource usage of MDSim is presumably very similar to a CPU micro benchmark. Conversely, the WPBench application is much more heterogeneous and its resource footprint is not inherently obvious using various kind of system resources. Beyond CPU-driven request processing, WPBench receives requests and sends responses over the network, reads and writes content from the file system, and requires the scheduler to switch between its various database or Web server processes. Therefore, it is not apparent whether micro benchmarks are able to capture such a varying workload. Nevertheless, the results revealed that the linear model is able to assess application response time surprisingly well, prevalently with error rates below 20%.

Benchmark	WPBench Read RT	WPBench Search RT	WPBench Write RT	MDSim Duration
Sysbench – CPU Multi Thread Duration				
RE±Range	12.5±7.1	17.5±8.7	40.8±34.9	8.2±4.7
R^2	99.2	98.9	42.5	99.8
Sysbench – CPU Single Thread Duration				
RE±Range	454±520	411.72±451	41.7±20.8	232±163
R^2	85.1	83.8	38.7	87.3
Baseline				
vCPUs				
RE±Range	616±607	546±515	127±89.8	317±184
R^2	68.0	68.7	28.1	68.3
ECU				
RE±Range	359±219	319±185.13	100±79.17	206±95
R^2	64.6	64.7	27.3	65.6
Max RE	2100	1810	140	600

Table 5.6: WPBench Response Time and MDSim Duration Estimators [%]

Benchmark	WPBench Read TP	WPBench Search TP	WPBench Write TP
StressNg – Network ICMP Ping IOPS			
RE±Range	23.0±27.5	24.8±27.4	31.1±16.9
R^2	66.4	57.4	25.6
Baseline			
vCPUs			
RE±Range	42.7±13.0	40.1±10.9	38.6±25.9
R^2	94.7	96.7	29.6
ECU			
RE±Range	27.0±30.5	27.3±25.8	30.6±19.4
R^2	91.5	87.8	30.1
Max RE	310	280	120

Table 5.7: WPBench Throughput Estimators [%]

Implications

Concurrency plays an important role when estimating the performance across instance types with a different number of vCPUs. The Sysbench – CPU single thread versus multi thread scenario revealed that micro benchmarks need to match its estimation target application in terms of optimization for multi core (*cf.*, multi vCPUs) platforms. It also shows that CPU micro benchmarks are suited to identify optimal instance types for workloads with a particular concurrency level (*e.g.*, single threaded). Further, it emphasizes that benchmark parameters, such as the level of concurrency, can have a profound impact on results.

The baseline metrics vCPU and ECU are insufficient to estimate the performance of certain applications. The number of vCPUs fails to capture fundamental technological differences such as different CPU clock frequencies and thus exhibits large relative errors for many instance types. Although the ECU metric yields considerably better estimates than vCPU, its relative error is still unacceptably high above 100%. Therefore, ECUs could be used at most to obtain a very rough estimate if no other metric is available but application specific micro benchmarks should be favored to obtain the most accurate application performance estimate. Finally, the number of vCPUs should never be used in isolation for estimating application performance.

Summary

The multi thread Sysbench – CPU benchmark serves as the best estimator for the response time of WPBench and the duration of MDSim. In all these cases, the benchmark-based performance estimates vastly outperform the baseline estimates using ECU or the number of vCPUs. For instance types with one and two vCPUs, the StressNg – Network ICMP Ping benchmark estimates the WPBench throughput best. However, its generally good fit is severely impacted by outliers and thus improvement upon the baseline is only marginal on average.

Chapter 6

Final Remarks

This chapters summarizes the contributions, concludes this thesis, and outlines future work.

6.1 Conclusion

This thesis investigated the relevancy of widely-used artificial micro benchmarks to estimate real-world application performance. A cloud benchmarking methodology has been designed that combines single-instance and multi-instance micro and application benchmarks. The methodology has been instantiated in a study with a market-leading cloud provider and a linear estimation model has been evaluated. Over 60000 measurements were collected to answer the research questions from Chapter 1:

RQ1 – Performance Variability *within Instance Types*

Does the performance of equally configured cloud instances vary relevantly?

Outcome: No. Performance does *not* vary relevantly for most benchmarks in Amazon's EC2 cloud for all intensively tested configurations in two different regions.

The low performance variability motivates inter-instance type performance estimation because only the sufficiency of small sample sizes makes such an approach practically viable:

RQ2 – Application Performance Estimation *across Instance Types*

Can a set of micro benchmarks estimate application performance for cloud instances of different configurations?

Outcome: Yes. Selective micro benchmarks are able to estimate certain application performance metrics with acceptable accuracy.

The sub-questions of RQ2 address the accuracy of the application performance estimates and the selection of suitable micro benchmark estimators:

RQ2.1 – Estimation Accuracy

How accurate can a set of micro benchmarks estimate application performance?

Outcome: A scientific computing application achieves relative error rates below 10% and the response time of a Web serving application is estimated with a relative error between 10% and 20%.

RQ2.2 – Micro Benchmark Selection

Which subset of micro benchmarks estimates application performance most accurately?

Outcome: A single CPU benchmark was able to estimate the duration of a scientific computing application and the response time of a Web serving application most accurately. It has also been shown that benchmarks cannot necessarily be used interchangeably even if they test the same resource and benchmark parameters can have a profound impact.

This thesis substantiates the suitability of micro benchmarks for estimating application performance but also highlights that only selected micro benchmarks are relevant regarding a particular application. Thus, this thesis motivates the use of such estimates during instance type selection as more insightful guidance compared to ordinal scale instance type rankings. It also emphasizes the importance of cloud benchmarking by showing that benchmark-based metrics can vastly improve estimation accuracy upon using instance specification-based metrics. Further, this thesis corroborates the dynamicity of cloud environments with indications that the tested cloud provider shifts from delivering best effort performance to specifically designed performance levels with high predictability.

6.2 Future Work

This section discusses possible extensions to this thesis and outlines a vision on how to reduce application profiling effort.

One non-addressed issue in this thesis is the threat to what extent the results are applicable to other cloud providers and application domains (*cf.*, threats to external validity Section 4.5.3). Beyond covering traditional instance types offered by well-known providers [Dor16], such as Microsoft Azure[1], a particularly interesting extension would examine individually tailorable instance types such as offered by Century Link[2] or Google's Cloud Platform[3]. Applications from other domains may find other suitable micro benchmark estimators or may reveal that the proposed linear regression model is insufficient to capture their miscellaneous performance bottlenecks.

The evaluation of the estimation model in this thesis is limited to single-instance applications. However, today's cloud environments are dominated by scale-out workload [FAK+12], where the

[1] `https://docs.microsoft.com/en-us/azure/virtual-machines/virtual-machines-windows-sizes`
[2] `https://www.ctl.io/servers/#Features`
[3] `https://cloud.google.com/custom-machine-types/`

components of an application are distributed across multiple instances. Therefore, a possible extension to apply the estimation model for multi-instance applications is outlined in the following. As a black-box approach, the estimation model is inherently incapable to optimize the sizing of individual instances for multi-instance applications. Nevertheless, estimates for individual application components can be combined by leveraging application-specific knowledge. For instance, given a throughput estimate for an application component behind a load balancer, the overall application throughput behaves asymptotically additive depending on the number of application components. Isolating individual application components remains a big challenge but can be alleviated by modular application architectures such as Microservices[4]. The estimation model is directly applicable to application components that can be isolated in the context of a VM. This implies that the load generator, external dependencies of the application component, and their inter-component network connections must not impose a bottleneck upon the application performance of interest. Thus, for application components without external dependencies (*e.g.*, the database in a typical three-tier Web application stack), the estimation model is directly applicable given the existence of a suitable load test. For application components with external dependencies (*e.g.*, the Web server has a dependency to the database), isolation can be partially simulated by intentionally over-provisioning external dependencies during the training phase of the model.

Currently, the estimation approach is evaluated in a traditional performance testing setting and primarily presented to support initial cloud instance selection. However, it is generally feasible to apply this approach to any kind of instance-type dependent application performance metrics. Therefore, future work can evaluate its accuracy with runtime performance metrics from different instance types to reduce dedicated performance testing efforts. Runtime metrics from performance monitoring tools such as Newrelic[5] can build a fine-grained application profile from representative real-world workloads for an instance type. Different instance types can be acquired to process production workload and simultaneously obtain training data. Starting from two training samples, the estimation approach can guide further instance type selection or reveal better offers from other providers. In this way, cloud instance selection can become an integral part of vertical scaling strategies instead of being perceived as wasted effort.

One key issue that limits the applicability of the proposed estimation approach in industrial settings is the need for application deployment and performance testing on at least two cloud instance types (*i.e.*, the boundary instances labeled as training data). Contemporary technology facilitates application deployment with the advent of container platforms such as Docker[6] or configuration management software such as Chef[7] or Puppet[8]. However, many applications still involve considerable manual labor to deploy and test in a cloud environment. Therefore, future work could explore whether VMM-based resource throttling (*e.g.*, CPU cap[9]) or tool-based resource limiting (*e.g.*, Cpulimit[10]) are able to simulate a wide range of instance types on a single machine. Such a visionary instance type simulator would allow to obtain training data for a large range of imaginary instance type configurations in a one-time effort without the need to port the application into the cloud. However, the training of the model in a different environment raises the big threat how representative such artificially introduced resource limits can capture real-world cloud resources. Notice that elaborate deployment is less relevant for micro benchmarks because they are typically easy to install and also generic. Thus, the continuous profiling effort across many instance types can be shared by a community or offered as a service by the cloud provider as recommended by Evangelinou *et al.* [ECA+16].

[4] `https://martinfowler.com/articles/microservices.html`
[5] `https://newrelic.com/`
[6] `https://www.docker.com/`
[7] `https://www.chef.io/chef/`
[8] `https://puppet.com/`
[9] `https://wiki.xen.org/wiki/Credit_Scheduler#Cap`
[10] `https://github.com/opsengine/cpulimit`

Appendix A

Abbreviations

AMI Amazon Machine Image

API Application Programming Interface

App Application

AWS Amazon Web Services

AZ Availability Zone

CDN Content Delivery Network

CMS Content Management System

CPU Central Processing Unit

CSV Comma-Separated Values

CWB Cloud WorkBench

DSL Domain Specific Language

EC2 Elastic Compute Cloud

ECU EC2 Compute Unit

EBS Elastic Block Storage

FIO Flexible I/O

GB Gigabyte (1 GB = 10^9 Bytes = 1 000 000 000 Bytes)

GHz Gigahertz

GiB Gibibyte (1 GiB = 2^{30} Bytes = 1 073 741 824 Bytes)

GPU Graphical Processing Unit

HDD Hard Disk Drive

HVM Hardware-assisted Virtual Machine

IaaS Infrastructure-as-a-Service

IaC Infrastructure as Code

ICMP Internet Control Message Protocol

IDE Integrated Development Environment

I/O Input/Output

IOPS Input/Output Operations per Second

IP Internet Protocol

KiB Kibibyte (1 KiB = 2^{10} Bytes = 1 024 Bytes)

KVM Kernel-based Virtual Machine

LAN Local Area Network

LQN Layered Queuing Network

MCT Multiple Consecutive Trials

MIT Multiple Interleaved Trials

MDSim Molecular Dynamics Simulation

NIST National Institute of Standards and Technology

OO Object Oriented

OLTP Online Transaction Processing

PaaS Platform-as-a-Service

PCC Pearson Correlation Coefficient

PV Para-Virtualization

RAM Random-Access Memory

RE Relative Error

REST Representational State Transfer

RMIT Randomized Multiple Interleaved Trials

RQ Research Question

RSD Relative Standard Deviation

RT Response Time

SaaS Software-as-a-Service

SD Standard Deviation

SSD Solid State Disk

SSH Secure Shell

SUT System Under Test

TCP Transmission Control Protocol

TP Throughput

UDP User Datagram Protocol

VM Virtual Machine

VMM Virtual Machine Monitor (*i.e.*, the hypervisor)

VPC Virtual Private Cloud

WPBench WordpressBench (*i.e.*, the Wordpress benchmark)

Bibliography

[AB17] Ali Abedi and Tim Brecht. Conducting repeatable experiments in highly variable cloud computing environments. In *8th ACM/SPEC International Conference on Performance Engineering (ICPE)*, April 2017.

[ACC+02] Cristiana Amza, Anupam Chanda, Alan L Cox, Sameh Elnikety, Romer Gil, Karthick Rajamani, Willy Zwaenepoel, Emmanuel Cecchet, and Julie Marguerite. Specification and implementation of dynamic web site benchmarks. In *Workload Characterization, 2002. WWC-5. 2002 IEEE International Workshop on*, pages 3–13. IEEE, 2002.

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, February 2009. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`.

[AI10] Syed A. Ahson and Mohammad Ilyas. *Cloud Computing and Software Services: Theory and Techniques*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2010. URL: `http://www.crcpress.com/product/isbn/9781439803158`.

[ALC+17] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 2017.

[BBG11] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud computing: Principles and Paradigms*, volume 87. John Wiley & Sons, March 2011. URL: `http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470887990.html`.

[BCDF10] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. arXiv preprint arXiv:1012.2599, 2010.

[BCX+06] Kevin J. Bowers, Edmond Chow, Huafeng Xu, Ron O. Dror, Michael P. Eastwood, Brent A. Gregersen, John L. Klepeis, Istvan Kolossvary, Mark A. Moraes, Federico D. Sacerdoti, John K. Salmon, Yibing Shan, and David E. Shaw. Scalable algorithms for molecular dynamics simulations on commodity clusters. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, 2006. `doi:10.1145/1188455.1188544`.

[BdDPP16] Alessio Botta, Walter de Donato, Valerio Persico, and Antonio Pescapé. Integration of cloud computing and internet of things: A survey. *Future Generation Computer Systems*, 56:684 – 700, 2016. doi:10.1016/j.future.2015.09.021.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003. doi:10.1145/1165389.945462.

[BGT12] Linda S. Fidell Barbara G. Tabachnick. *Using Multivariate Statistics*. 6th Edition. Pearson, 6 edition, 2012.

[BKKL09] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the weather tomorrow?: Towards a benchmark for the cloud. In *Proceedings of the Second International Workshop on Testing Database Systems (DBTest '09)*, pages 9:1–9:6. ETH Zurich, 2009. doi:10.1145/1594156.1594168.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3 – 22, 2009. Special Issue: Software Performance - Modeling and Analysis. doi:10.1016/j.jss.2008.03.066.

[BLL+14] A. H. Borhani, P. Leitner, B. S. Lee, X. Li, and T. Hung. Wpress: An application-driven performance benchmark for cloud-based virtual machines. In *2014 IEEE 18th International Enterprise Distributed Object Computing Conference*, pages 101–109, Sept 2014. doi:10.1109/EDOC.2014.23.

[BNC+16] Dave Bartoletti, Lauren E. Nelson, Andras Cser, Sophia I. Vargas, William Martorelli, Liz Herbert, Andre Kindness, Paul Miller, Charlie Dai, and Frank Liu. Predictions 2017: Customer-obsessed enterprises launch cloud's second decade, November 2016.

[BS10] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems (MMSys '10)*, pages 35–46, 2010. doi:10.1145/1730836.1730842.

[BYV+09] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009. doi:10.1016/j.future.2008.12.001.

[CBMG16] Mauro Canuto, Raimon Bosch, Mario Macias, and Jordi Guitart. A methodology for full-system power modeling in heterogeneous data centers. In *Proceedings of the 9th International Conference on Utility and Cloud Computing (UCC '16)*, pages 20–29, 2016. doi:10.1145/2996890.2996899.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008. doi:10.1145/1365815.1365816.

[CGPS12] D. Cerotti, M. Gribaudo, P. Piazzolla, and G. Serazzi. Flexible cpu provisioning in clouds: A new source of performance unpredictability. In *2012 Ninth International Conference on Quantitative Evaluation of Systems*, pages 230–237, Sept 2012. doi:10.1109/QEST.2012.23.

[CMS13] Matheus Cunha, Nabor Mendonça, and Américo Sampaio. A declarative environment for automatic performance evaluation in iaas clouds. In *Sixth IEEE International Conference on Cloud Computing (CLOUD)*, pages 285–292, June 2013. doi:10.1109/CLOUD.2013.12.

[Con16] The SPEC Consortium. Spec cloud™ iaas 2016 benchmark, 2016. URL: http://spec.org/cloud_iaas2016/ [cited 2016-09-08].

[Dav16] Christian Davatz. Who provides the most bang for the buck? an application-benchmark based performance analysis of two iaas providers. Master's thesis, University of Zurich – Software Evolution and Architecture Lab s.e.a.l., August 2016.

[DISL17] Christian Davatz, Christian Inzinger, Joel Scheuner, and Philipp Leitner. An approach and case study of cloud instance type selection for multi-tier web applications. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2017.

[Dor16] Lydia Leong; Gregor Petri; Bob Gill; Mike Dorosh. Magic quadrant for cloud infrastructure as a service, worldwide, August 2016. URL: https://www.gartner.com/doc/reprints?id=1-2G2O5FC&ct=150519.

[DPC10] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. *EC2 Performance Analysis for Resource Provisioning of Service-Oriented Applications*, pages 197–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-16132-2_19.

[ECA+16] Athanasia Evangelinou, Michele Ciavotta, Danilo Ardagna, Aliki Kopaneli, George Kousiouris, and Theodora Varvarigou. Enterprise applications cloud rightsizing through a joint benchmarking and optimization approach. *Future Generation Computer Systems*, pages –, 2016. doi:10.1016/j.future.2016.11.002.

[EGHO16] Rania El-Gazzar, Eli Hustad, and Dag H. Olsen. Understanding cloud computing adoption issues: A delphi study approach. *Journal of Systems and Software*, 118:64 – 84, 2016. doi:https://doi.org/10.1016/j.jss.2016.04.061.

[EKKJP10] Y. El-Khamra, H. Kim, S. Jha, and M. Parashar. Exploring the performance fluctuations of hpc workloads on clouds. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 383–387, Nov 2010. doi:10.1109/CloudCom.2010.84.

[FAK+12] 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, 2012. doi:10.1145/2150976.2150982.

[FAS+13] Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. Benchmarking in the cloud: What it should, can, and cannot be. In *Selected Topics in Performance Evaluation and Benchmarking*, volume 7755 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2013. doi:10.1007/978-3-642-36727-4_12.

[FE04] Ian Foster and Carl Kesselman (Eds.). *The Grid 2. Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2 edition, 2004.

[FJV$^+$12] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*, pages 20:1–20:14, 2012. doi:10.1145/2391229.2391249.

[FvdM16] Amy Ann Forni and Rob van der Meulen. Gartner says by 2020, a corporate "no-cloud" policy will be as rare as a "no-internet" policy is today, June 2016. URL: http://www.gartner.com/newsroom/id/3354117.

[FZRL08] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop*, pages 1–10, Nov 2008. doi:10.1109/GCE.2008.4738445.

[Gar07] Simson L. Garfinkel. An evaluation of amazon's grid computing services: Ec2, s3 and sqs. Technical report, Center for, 2007.

[Gre13] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Prentice Hall, 2013.

[Ham09] James Hamilton. Cooperative expendable micro-slice servers (cems): low cost, low power servers for internet-scale services. In *Conference on Innovative Data Systems Research (CIDR'09)(January 2009)*, 2009.

[Hil09] David Hilley. Cloud computing: A taxonomy of platform and infrastructure-level offerings. Technical Report GIT-CERCS-09-13, Georgia Institute of Technology, 2009. URL: http://www.cercs.gatech.edu/tech-reports/tr2009/git-cercs-09-13.pdf.

[HK10] Christina N. Höfer and Georgios Karagiannis. Taxonomy of cloud computing services. In *IEEE Globecom Workshops*, pages 1345–1350, December 2010. doi:10.1109/GLOCOMW.2010.5700157.

[HPE$^+$06] Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John, and Koen De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT '06)*, pages 114–122, 2006. doi:10.1145/1152154.1152174.

[IOY$^+$11] Alexandru Iosup, Simon Ostermann, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):931–945, June 2011. doi:10.1109/TPDS.2011.66.

[IYE11] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. On the performance variability of production cloud services. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 104–113, May 2011. doi:10.1109/CCGrid.2011.22.

[JKC$^+$13] Deepal Jayasinghe, Josh Kimball, Siddharth Choudhary, Tao Zhu, and Calton Pu. An automated approach to create, store, and analyze large-scale experimental data in clouds. In *14th IEEE International Conference on Information Reuse and Integration (IRI)*, pages 357–364, August 2013. doi:10.1109/IRI.2013.6642493.

[JSM+12] Deepal Jayasinghe, Galen Swint, Simon Malkowski, Jack Li, Qingyang Wang, Junhee Park, and Calton Pu. Expertus: A generator approach to automate performance testing in iaas clouds. In *5th IEEE International Conference on Cloud Computing (CLOUD)*, pages 115–122, June 2012. `doi:10.1109/CLOUD.2012.98`.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[Kot14] Lars Kotthoff. Reliability of computational experiments on virtualised hardware. *Journal of Experimental & Theoretical Artificial Intelligence*, 26(1):33–49, 2014. `doi:10.1080/0952813X.2013.784812`.

[KPP+02] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[KSHD13] Steffen Kächele, Christian Spann, Franz J. Hauck, and Jörg Domaschka. Beyond iaas and paas: An extended cloud taxonomy for computation, storage and networking. In *6th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pages 75–82, December 2013. `doi:10.1109/UCC.2013.28`.

[LC16] Philipp Leitner and Jürgen Cito. Patterns in the chaos — a study of performance variation and predictability in public iaas clouds. *ACM Trans. Internet Technol.*, 16(3):15:1–15:23, April 2016. `doi:10.1145/2885497`.

[LOZC12] Zheng Li, Liam O'Brien, He Zhang, and Rainbow Cai. On a catalogue of metrics for evaluating commercial cloud services. In *Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing (GRID '12)*, pages 164–173, 2012. `doi:10.1109/Grid.2012.15`.

[LS15] Philipp Leitner and Joel Scheuner. Bursting With Possibilities – an Empirical Study of Credit-Based Bursting Cloud Instance Types. In *8th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2015. `doi:10.1109/UCC.2015.39`.

[LYKZ10] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: Comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10)*, pages 1–14, 2010. `doi:10.1145/1879141.1879143`.

[LZK+11] Ang Li, Xuanran Zong, Srikanth Kandula, Xiaowei Yang, and Ming Zhang. Cloudprophet: Towards application performance prediction in cloud. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*, pages 426–427, 2011. `doi:10.1145/2018436.2018502`.

[LZZ+11] Ang Li, Xuanran Zong, Ming Zhang, Srikanth Kandula, and Xiaowei Yang. Cloudprophet: predicting web application performance in the cloud. *ACM SIGCOMM Poster*, 2011.

[Mag09] Frederic Magoules. *Fundamentals of Grid Computing: Theory, Algorithms and Technologies*. Chapman & Hall/CRC, 1st edition, 2009.

[MG11] Peter Mell and Timothy Grance. The nist definition of cloud computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), September 2011. URL: `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`.

[Nad14] Satya Nadella. Mobile first, cloud first [online]. March 2014. Press Briefing of Microsoft CEO. URL: `http://www.microsoft.com/en-us/news/speeches/2014/03-27nadella.aspx` [cited 2014-07-07].

[OIY+10] Simon Ostermann, Alexandria Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *Cloud Computing*, volume 34 of *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, pages 115–131. Springer, 2010. `doi:10.1007/978-3-642-12636-9_9`.

[OZL+13] Z. Ou, H. Zhuang, A. Lukyanenko, J. K. Nurminen, P. Hui, V. Mazalov, and A. Ylä-Jääski. Is the same instance type created equal? exploiting heterogeneity of public clouds. *IEEE Transactions on Cloud Computing*, 1(2):201–214, July 2013. `doi:10.1109/TCC.2013.12`.

[OZN+12] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. Exploiting hardware heterogeneity within the same instance type of amazon ec2. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'12)*, 2012. URL: `http://dl.acm.org/citation.cfm?id=2342763.2342767`.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974. `doi:10.1145/361011.361073`.

[PG17] Christy Pettey and Laurence Goasduff. Gartner says worldwide public cloud services market to grow 18 percent in 2017, February 2017. URL: `http://www.gartner.com/newsroom/id/3616417`.

[Por16] Matthew Portnoy. *Virtualization Essentials*. Sybex, 2 edition, 2016.

[PSF16] Tapti Palit, Yongming Shen, and Michael Ferdman. Demystifying cloud benchmarking. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 122–132, April 2016.

[RS95] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, Aug 1995. `doi:10.1109/32.403785`.

[Sch14] Joel Scheuner. Cloud WorkBench. Bachelor's thesis, University of Zurich – Software Evolution and Architecture Lab s.e.a.l., 2014.

[SCLG15] Joel Scheuner, Jürgen Cito, Philipp Leitner, and Harald Gall. Cloud WorkBench: Benchmarking IaaS Providers based on Infrastructure-as-Code. In *Proceedings of the 24th International World Wide Web Conference (WWW'15) - Demo Track*, 2015.

[SDQR10] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3(1):460–471, September 2010. `doi:10.14778/1920841.1920902`.

[SHG+13] M. Silva, M.R. Hines, D. Gallo, Qi Liu, Kyung Dong Ryu, and D. Da Silva. Cloudbench: Experiment automation for cloud environments. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 302–311, March 2013. `doi:10.1109/IC2E.2013.33`.

[SLCG14] Joel Scheuner, Philipp Leitner, Jürgen Cito, and Harald Gall. Cloud WorkBench - Infrastructure-as-Code Based Cloud Benchmarking. In *Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'14)*, 2014. doi:10.1145/2740908.2742833.

[Spe17] Cloud Spectator. Price-performance analysis of the top 10 public iaas vendors. Technical report, Cloud Spectator, 2017.

[SS05] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 71–84, Berkeley, CA, USA, 2005. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1251203.1251209.

[SSS+08] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, Armando Fox, and David Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, 2008.

[VAM+14] Blesson Varghese, Ozgur Akgun, Ian Miguel, Long Thai, and Adam Barker. Cloud benchmarking for performance. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 535–540. IEEE, 2014.

[VAM+16] Blesson Varghese, Ozgur Akgun, Ian Miguel, Long Thai, and Adam Barker. Cloud benchmarking for maximising performance of scientific applications. *arXiv preprint arXiv:1608.00406*, 2016.

[VRMCL08] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: Towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, January 2008. doi:10.1145/1496091.1496100.

[VSTB16] Blesson Varghese, Lawan Thamsuhang Subba, Long Thai, and Adam Barker. Container-based cloud virtual machine benchmarking. *arXiv preprint arXiv:1601.03872*, 2016.

[Wal08] Edward Walker. Benchmarking amazon ec2 for high-performance scientific computing. *Usenix Login*, 33(5):18–23, October 2008.

[WKP10] Hyrum K. Wright, Miryung Kim, and Dewayne E. Perry. Validity concerns in software engineering research. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*, pages 411–414, 2010. doi:10.1145/1882362.1882446.

[WLJ+16] Nianxin Wang, Huigang Liang, Yu Jia, Shilun Ge, Yajiong Xue, and Zhining Wang. Cloud computing research in the {IS} discipline: A citation/co-citation analysis. *Decision Support Systems*, 86:35 – 47, 2016. doi:10.1016/j.dss.2016.03.006.

[WN10] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon ec2 data center. In *Proceedings IEEE INFOCOM*, pages 1–9, March 2010. doi:10.1109/INFCOM.2010.5461931.

[XMNB13] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *NSDI*, volume 13, pages 329–342, 2013.

[Yin08] Robert K Yin. *Case Study Research: Design and Methods*. Applied Social Research Methods. SAGE Publications, 4th edition, 2008.

[You17] M. Yousif. The state of the cloud. *IEEE Cloud Computing*, 4(1):4–5, Jan 2017. doi: 10.1109/MCC.2017.4.

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010. doi: 10.1007/s13174-010-0007-6.

[ZL11] Gong Zhang and Ling Liu. Why do migrations fail and what can we do about it? In *Proceedings of the 25th International Conference on Large Installation System Administration (LISA'11)*, pages 25–25, 2011.