Modeling and Improving the Performance of Cloud Systems

by

Panagiotis (Panos) Patros

MCS, University of New Brunswick, 2014
BSc in Informatics and Telecommunications, National and Kapodistrian University of Athens, 2010

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

In the Graduate Academic Unit of Computer Science

Supervisor(s): Kenneth Kent, PhD Computer Science
Examinining Board: Eric Aubanel, PhD, Computer Science, Chair
Rainer Herpers, PhD, Computer Science
Suprio Ray, PhD, Computer Science
Maryhelen Stevenson, PhD, Electrical and Computer Engineering

External Examiner: Cristiana Amza, PhD, Electrical and Computer Engineering, University of Toronto

This dissertation is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

June, 2018

©Panagiotis (Panos) Patros, 2018
Abstract

Cloud computing enables the transparent allocation and sharing of computing resources offered by a multitenant cloud provider to its tenants. Cloud applications are predominately following the client-server paradigm; the server part of an application is the one executed on the cloud and it is responsible for serving its clients while abiding by certain constraints, called Service Level Agreements (SLAs). However, multitenancy comes at the cost of performance interference, which can risk SLA satisfaction. This work investigates a number of performance improvements targeting multitenant cloud application servers as well as theoretically modeling and predicting their behavior. In particular, multitenancy at the application server level is proposed and evaluated; CPU contention during scaling is modeled and cloud applications are classified based on their resource slowdown; cloud performance during scaling is improved through a cross-instance experience sharing technique; the SLA satisfaction of tenants is modeled and improved; the effects of Garbage Collection on SLA satisfaction are studied; and low-load periods are utilized towards decreasing GC-induced performance interference.
Dedication

To Bertrand Russell.
Acknowledgements

The following people have had a significant impact, either directly or indirectly, in the completion of this thesis:


Thank you!
# Table of Contents

Abstract ii  
Dedication iii  
Acknowledgments iv  
Table of Contents v  
List of Tables xi  
List of Figures xii  
Table of Symbols, Nomenclature or Abbreviations xxi  
1 Introduction 1  
2 Background 6  
2.1 Operating Systems 8  
2.1.1 Linux Control Groups 9  
2.1.2 Virtualization 10  
2.2 Cloud Computing 11
7.5.5 GC Policies Discussion ........................................ 162
7.6 Conclusion ......................................................... 165

8 Mitigating GC Interference ....................................... 167
  8.1 Motivation ....................................................... 168
  8.2 Related Work .................................................. 169
  8.3 Elastic GC ....................................................... 171
    8.3.1 Design and Implementation .............................. 171
  8.4 Experimental Evaluation ................................... 173
    8.4.1 Discussion ................................................ 175
    8.4.2 Resource Consumption Investigation .................. 177
  8.5 Conclusions ................................................... 180

9 Conclusions ......................................................... 183
  9.1 Future Directions ............................................. 187

Bibliography ........................................................... 212

Vita
List of Tables

4.1 The experimental results of the resource consumption of Cloud Burners reveal that each burner stressed its intended resource more than its counterparts. ........................................ 47
4.2 Description of the Java EE cloud-oriented applications implemented for testing purposes. .................................................. 52
6.1 Root mean square error calculations between theoretically predicted and experimentally acquired results that illustrate the efficacy of the proposed model. ........................................ 113
6.2 Aggregate experimental results grouped by various numbers of parallel clients. Green shading means improvement; whereas, red means worsening in comparison to the baseline. ........... 123
7.1 GC Policies of the IBM J9 JVM, which runs on OMR, and their corresponding features. .................................................. 130
7.2 The configuration parameters and presets of CloudGC. ........ 143
7.3 List of dependent variables and experimental metrics captured while experimenting with CloudGC. ......................... 148
List of Figures

1.1 Flow of the research chapters: Chapter 3 proposes multitenancy at the application server. This came at a cost of reduced performance; thus, Chapter 4 investigates the effect of multitenancy interference in performance and scaling. A technique to mitigate performance interference and reduce startup times in cloud applications is presented in Chapter 5; a multitenant-load aware reordering technique is presented in Chapter 6. Garbage Collection effects in multitenancy became apparent; their impacts are studied in Chapter 7 and mitigated in Chapter 8. ................................................................. 3

2.1 The contemporary Internet (of Things) connects multiple types of devices to each other; clouds frequently act as a centralized point of contact for communication, processing and storage purposes. ................................................................. 7

2.2 A simplified operating system architecture exposing the underlying hardware resources and managing their provision to multiple users and applications. .............................. 8
2.3 Cloud computing service models include IaaS, PaaS and SaaS. Each abstract and provide increasingly larger pieces of the hardware/software stack to their clients; they also tend to hierarchically leverage each other.

2.4 A sample architecture of a PaaS cloud system that manages a number of hosts and deploys user containers on them. PaaS containers commonly implement large parts of the software stack required by the user.

2.5 Language runtimes split their memory into stack and heap. Data on the stack is available to the program depending on the running function’s scope. Objects are allocated on the heap and are alive when they are accessible through a chain of links from the stack; otherwise, they are dead.

3.1 High-level memory sharing architecture of the ASaaS multi-tenant technique on an example enterprise stack.

3.2 The memory models of the ASaaS mode versus the baseline standard and RaaS modes.

3.3 Average response time speedup (top) and memory footprint (bottom) for Hello World tenants.

3.4 Average response time speedup (top) and memory footprint (bottom) for DB-Intensive tenants.
3.5 Comparison of resident memory set sizes for DB-Intensive tenants of the three tested modes for various numbers of deployed tenants. .......................................................... 38

4.1 Number of cache misses caused by the cache burner for a varying number of parallel threads. .................................................. 48

4.2 Number of cache misses caused by the cache burner and its resident memory set size for varying buffer sizes its threads used. 49

4.3 Number of cache misses caused by the memory burner and its resident memory set size for a varying number of parallel threads. .................................................. 49

4.4 The experimentally acquired resource slowdown vectors of the tested applications are displayed with the use of radar charts. The lower the value on a specific resource, the slower the application became in the presence of the corresponding cloud burner. .................................................. 54

4.5 Experimental scaling results on a local and isolated Cloud Foundry installation. .................................................. 65

4.6 Theoretically predicted CPU allocation using the proposed model. .................................................. 67

4.7 Experimentally measured throughput for the LongDoubleSums Java EE application, which resembles well the theoretical CPU utilization predictions of the model .................................................. 68

xiv
4.8 Experimentally measured throughput for the DoubleArray-
Sum Java EE application, which resembles the theoretical
CPU utilization predictions of the model. 69

4.9 Experimental speedup results of scaling the testing applica-
tions on the local and isolated Cloud Foundry installation with
the CPU burner activated. 70

4.10 Experimentally measured throughput scaling speedups on a
commercial cloud that runs Cloud Foundry (top 32 clients,
bottom 64 clients). 71

5.1 A sequence chart outlining the various interactions in the pro-
posed Dynamically Compiled Artifact Sharing (DCAS) tech-
nique. 83

5.2 Experimentally measured scale-out performance on a local and
isolated Cloud Foundry installation comparing DCAS against
the baseline. 87

5.3 Experimentally measured throughput interference comparison
between DCAS and the default mode that is imposed on an-
other tenant running on the same local and isolated installa-
tion of Cloud Foundry. 89
5.4 Experimentally measured response time interference—top for 90th and bottom for 99th percentile—comparison between DCAS and the default mode that is imposed on another tenant running on the same local and isolated installation of Cloud Foundry. .................................................. 90

5.5 Experimentally measured hashing time comparisons of various parts of the Liberty buildback container using the \texttt{md5sum} algorithm for four different Java EE applications. ................. 91

6.1 Connection of on-time and percentile metrics with a distribution graph of the response time. The shaded area is the theoretical on-time percentage; a certain quotient corresponds to the theoretical response time percentile. ......................... 102

6.2 Comparison of the theoretical model predictions on the left versus the experimentally acquired measurements run on a local and isolated Cloud Foundry installation on the right. (Part 1/2). .......................................................... 103

6.3 Comparison of the theoretical model predictions on the left versus the experimentally acquired measurements run on a local and isolated Cloud Foundry installation on the right. (Part 2/2). .......................................................... 104
6.4 Comparison of theoretical and experimentally acquired results (run on a local and isolated installation of Cloud Foundry) of maintaining the OnTime% above a certain SLO threshold for a varying number of parallel clients and $\mu = 0.5, \sigma = 0.25$ . . . 110

6.5 Comparison of theoretical and experimentally acquired results (run on a local and isolated installation of Cloud Foundry) of maintaining the OnTime% above a certain SLO threshold for a varying number of parallel clients and $\mu = 0.25, \sigma = 0.25$ . . . 111

6.6 Comparison of theoretical and experimentally acquired results (run on a local and isolated installation of Cloud Foundry) of maintaining the OnTime% above a certain SLO threshold for a varying number of parallel clients and $\mu = 0.5, \sigma = 0.1$ . . . 112

6.7 Two diagrams illustrating the proposed algorithm of SLO request reordering. The diagram on the left describes the actions to be performed when a new request arrives on the applications server; whereas, the diagram on the right, outlines the algorithm of the threads in the SLA pool. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114

6.8 Experimentally acquired results on a local and isolated installation of Cloud Foundry comparing the performance of the proposed SLO mode against the baseline. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
6.9 Experimentally acquired results on a local and isolated installation of Cloud Foundry comparing the performance of the proposed SLO mode against the discussed Long-First and Short-First modes.

7.1 CloudGC organizes its memory in a stack frame architecture to replicate the way real-life applications work. Essentially, this design and based on the configuration parameters enables the existence of cold long-living objects, hot short-living objects, objects dying en-masse, objects escaping their allocation thread, etc.

7.2 Each CloudGC object maintains a number of pointer slots that can either point to other objects or contain null—represented with 0 in the diagram. Additionally, CloudGC objects contain a byte array for payload, which apart from enlarging the object, it can be used for read and writes that pollute the CPU cache without altering the object graph.

7.3 Graphs displaying the cumulative mass of response times. The x-axes describe the response time in ms; whereas the y-axes denote the cumulative mass of the response time in thousands of samples. The top nine show the non-interference experiments, the bottom nine, the ones with interference enabled.
7.4 Experimentally measured number of GCs per request. The top nine show the non-interference experiments, the bottom nine, the ones with interference enabled. .......................... 154

7.5 Experimentally measured number of GC time per request. The top nine show the non-interference experiments, the bottom nine, the ones with interference enabled. .......................... 155

7.6 Experimentally measured throughput reductions of the background interfering C++ tenant per different combination of GC policy, preset and number of parallel clients. .......................... 160

7.7 Experimentally measured response times increases of the background interfering C++ tenant per different combination of GC policy, preset and number of parallel clients. .......................... 161

8.1 Overall algorithm of the proposed Elastic GC technique that scales the runtime’s resources aiming to minimize interference on co-located tenants. .................................. 172

8.2 Experimentally acquired reductions of a co-located C++ tenant’s SLO violations with Elastic GC (Top, CloudGC Setting 1; Middle, CloudGC Setting 2; Bottom, CloudGC Setting 3) (Left, Response time; Right, Throughput). .......................... 175

8.3 Experimentally acquired overhead measurements of the Elastic GC algorithm on the foreground Java EE tenant. .......................... 176
8.4 Experimentally acquired C++ tenant’s throughput (left) and response time (right) measurements of a sample run to illustrate the differences caused by Elastic GC over the default. 178

8.5 Experimentally acquired CPU utilization measurements of one sample run of the foreground Java EE tenant reveal that Elastic GC caps the spikes during periods of low loads in comparison to the default. 179

8.6 Experimentally acquired aggregate throughput (left) and response time (right) of the C++ tenant during periods of interference. 179

8.7 Experimentally acquired aggregate CPU utilization and resident memory set size of the Java EE tenant. 180
# Table of Symbols,

## Nomenclature or Abbreviations

<table>
<thead>
<tr>
<th>Term</th>
<th>Full Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>≈</td>
<td>Approximately</td>
<td>A mathematical operator denoting that two parts are roughly equal to each other.</td>
</tr>
<tr>
<td></td>
<td>Equals</td>
<td></td>
</tr>
<tr>
<td>∃</td>
<td>Exists</td>
<td>A logical operation describing a preposition for at least one element in a set.</td>
</tr>
<tr>
<td>∀</td>
<td>For all</td>
<td>A logical operation describing a preposition for every element in a set.</td>
</tr>
<tr>
<td>≫</td>
<td>Much Greater Than</td>
<td>A mathematical operator denoting that the left part is much greater in value than the right.</td>
</tr>
<tr>
<td>∫</td>
<td>Integral</td>
<td>The area between a function and the x-axis.</td>
</tr>
<tr>
<td>⇒</td>
<td>Implies</td>
<td>A logical operation describing that the right part must be true, if the left part is true.</td>
</tr>
<tr>
<td>Term</td>
<td>Full Name</td>
<td>Explanation</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>∑</td>
<td>Sum</td>
<td>A mathematical operator denoting the sum of the contained sequence.</td>
</tr>
<tr>
<td>aaaS</td>
<td>as a Service</td>
<td>Some service offered in the pay-as-you-go cloud model.</td>
</tr>
<tr>
<td>AOT</td>
<td>Ahead Of Time</td>
<td>An operation executed before it is required. In this thesis, it is used in AOT compiling, the process of compiling bytecode to machine code by a language runtime before the user needs it.</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
<td>A set of specifications exposed for programming purposes.</td>
</tr>
<tr>
<td>ASaaS</td>
<td>Application Server as a Service</td>
<td>A proposed cloud model that offers application server services.</td>
</tr>
<tr>
<td>b</td>
<td>Bit</td>
<td>The most fundamental storage unit. It is binary and can store either a '1' or a '0'.</td>
</tr>
<tr>
<td>B</td>
<td>Byte</td>
<td>A fundamental storage unit comprised of 8 bits.</td>
</tr>
<tr>
<td>Balanced</td>
<td>Balanced</td>
<td>A region-based garbage collection policy of the IBM JVM that splits the heap to multiple independent chunks that can be collected in groups and aiming in minimizing maximum pauses.</td>
</tr>
<tr>
<td>CF</td>
<td>Cloud Foundry</td>
<td>A popular open source PaaS software.</td>
</tr>
<tr>
<td><strong>Term</strong></td>
<td><strong>Full Name</strong></td>
<td><strong>Explanation</strong></td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>----------------</td>
</tr>
<tr>
<td>cgroup</td>
<td>Linux Control Groups</td>
<td>The Linux Operating System offers cgroups for managing resources and isolating filesystems.</td>
</tr>
<tr>
<td>CP</td>
<td>Class Path</td>
<td>A set of directories that are scanned by the Java runtime to find compiled Java code.</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
<td>A computer hardware part that executes instructions.</td>
</tr>
<tr>
<td>CPU%</td>
<td>Central Processing Unit Utilization Percentage</td>
<td>The proportion of CPU cycles a process occupies on a given time window.</td>
</tr>
<tr>
<td>DB</td>
<td>Database</td>
<td>An application that persistently stores data as well as offering relevant services such as indexes for fast retrieval.</td>
</tr>
<tr>
<td>DCAS</td>
<td>Dynamically Compiled Artifact Sharing</td>
<td>A proposed cloud technique for improving startup times and reducing interference.</td>
</tr>
<tr>
<td>DEA</td>
<td>Droplet Execution Agent</td>
<td>A subsystem VM of the Cloud Foundry PaaS software that is responsible for spinning up containers that run the users’ code.</td>
</tr>
<tr>
<td>ds</td>
<td>Decisecound</td>
<td>A time unit equal to $10^{-1}$ seconds.</td>
</tr>
<tr>
<td>EAR</td>
<td>Enterprise Archive</td>
<td>A file containing compiled Java EE code and settings for deployment on an application server.</td>
</tr>
<tr>
<td>Term</td>
<td>Full Name</td>
<td>Explanation</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FIFO or</td>
<td>First In First Out or First Come</td>
<td>A data structure or methodology that maintains objects in a queue and processes the elements in order of arrival.</td>
</tr>
<tr>
<td>FCFS</td>
<td>First Serve</td>
<td></td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
<td>A (re)programmable circuit that can emulate other circuits.</td>
</tr>
<tr>
<td>GB</td>
<td>Gigabyte</td>
<td>A storage unit equal to 1,024 megabytes or 1,024³ bytes.</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collection</td>
<td>An algorithm that automatically finds and releases unusable memory.</td>
</tr>
<tr>
<td>GenCon</td>
<td>Generational Concurrent</td>
<td>The default garbage collection policy of the IBM JVM. It aims in utilizing the generational “most-objects-die-young” hypothesis by performing frequent collections on recently allocated objects.</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose Graphics</td>
<td>A highly parallelized processing unit following the single instruction multiple data paradigm.</td>
</tr>
<tr>
<td></td>
<td>Processing Unit</td>
<td></td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
<td>An application module that interacts with a human user via a set of visual input and output elements.</td>
</tr>
<tr>
<td>Term</td>
<td>Full Name</td>
<td>Explanation</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transfer Protocol</td>
<td>An application protocol for submitting information over the web. Internet browsers on the client-side and application/web server on the server-side implement it.</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Secure Hyper Text Transfer Protocol</td>
<td>An encrypted version of HTTP for secure transmission.</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
<td>Devices or data buses that connect a computer to the outside world.</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service</td>
<td>Lower level cloud model that offers hardware, virtual or physical hosts, power, networking, etc.</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines</td>
<td>A major international company focusing on various aspects of software engineering including cloud computing and language runtimes.</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
<td>The set of interconnected devices over the network, focusing on embedded systems.</td>
</tr>
<tr>
<td>J9</td>
<td>The IBM J9 JVM</td>
<td>IBM’s Java runtime.</td>
</tr>
<tr>
<td>Java EE</td>
<td>Java Enterprise Edition</td>
<td>A set of specifications in Java to support web-based services. Java EE applications require an appropriate application server that implements the EE spec in addition to a Java runtime.</td>
</tr>
<tr>
<td>Term</td>
<td>Full Name</td>
<td>Explanation</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Java SE</td>
<td>Java Standard Edition</td>
<td>The basic version of Java that focuses on console applications. Java SE applications require a Java runtime.</td>
</tr>
<tr>
<td>JCL</td>
<td>Java Class Library</td>
<td>A set of Java community classes that provide frequently used functionality.</td>
</tr>
<tr>
<td>JIT</td>
<td>Just In Time</td>
<td>An operation that is deferred as much as possible. In this thesis, it is used in JIT compiling, which is the process of compiling bytecode to machine code by a language runtime while the user’s code is running.</td>
</tr>
<tr>
<td>JSP</td>
<td>Java Server Page</td>
<td>A Java EE technology for creating dynamic content websites.</td>
</tr>
<tr>
<td>JSR</td>
<td>Java Specification Request</td>
<td>A software engineering document outlined desired changes in Java and maintained by its community.</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
<td>A language runtime that interprets Java bytecode.</td>
</tr>
<tr>
<td>KB</td>
<td>Kilobyte</td>
<td>A storage unit equal to 1,024 bytes.</td>
</tr>
<tr>
<td>MB</td>
<td>Megabyte</td>
<td>A storage unit equal to 1,024 kilobytes or $1,024^2$ bytes.</td>
</tr>
<tr>
<td>ms</td>
<td>Millisecond</td>
<td>A time unit equal to $10^{-3}$ seconds.</td>
</tr>
<tr>
<td>Term</td>
<td>Full Name</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Opt</td>
<td>Optimized</td>
<td>A garbage collection policy of the IBM JVM that aims in minimizing average</td>
</tr>
<tr>
<td>Avg-Pause</td>
<td>Average Pause</td>
<td>pause times by marking objects concurrently with the user threads.</td>
</tr>
<tr>
<td>Opt</td>
<td>Optimized</td>
<td>A garbage collection policy of the IBM JVM that aims in minimizing</td>
</tr>
<tr>
<td>Thruput</td>
<td>Throughput</td>
<td>collection times by deferring them as much as possible.</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
<td>A system’s piece of software that connects user and application to the</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform as a</td>
<td>underlying computer hardware.</td>
</tr>
<tr>
<td></td>
<td>Service</td>
<td>Middle level cloud model that offers large parts of the hardware/software</td>
</tr>
<tr>
<td>PHP</td>
<td>PHP: Hypertext</td>
<td>A server-side scripting language with a recursive acronym.</td>
</tr>
<tr>
<td></td>
<td>Preprocessor</td>
<td></td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
<td>The overall performance experienced by the end user relevant to their</td>
</tr>
<tr>
<td>RaaS</td>
<td>Runtime as a</td>
<td>A proposed cloud model that offers language runtime services.</td>
</tr>
<tr>
<td></td>
<td>Service</td>
<td></td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access</td>
<td>A type of memory that allows elements to be accessed directly without</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>having to traverse other elements. Computers’ main memories are normally</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RAM.</td>
</tr>
<tr>
<td>Term</td>
<td>Full Name</td>
<td>Explanation</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RIV</td>
<td>Resource Intensiveness</td>
<td>A proposed cloud multitenancy metric that describes the magnitude an application speeds up in the absence of a specific resource-intensive tenant per dimension.</td>
</tr>
<tr>
<td></td>
<td>Vector</td>
<td></td>
</tr>
<tr>
<td>RSS</td>
<td>Resident Set Size</td>
<td>The number of bytes a process occupies in the main memory.</td>
</tr>
<tr>
<td>RSV</td>
<td>Resource Slowdown Vector</td>
<td>A proposed cloud multitenancy metric that describes the magnitude an application slows down in the presence of a specific resource-intensive tenant per dimension.</td>
</tr>
<tr>
<td>s</td>
<td>Second</td>
<td>A fundamental measurement unit of time.</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
<td>Higher level cloud model that offers applications.</td>
</tr>
<tr>
<td>SCC</td>
<td>Shared Class Cache</td>
<td>An IBM JVM technique for enabling the sharing of read-only Java class components among runtimes over a shared address space.</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
<td>Set of contractual obligations agreed between a cloud provider and a client.</td>
</tr>
<tr>
<td>SLI</td>
<td>Service Level Indicator</td>
<td>A performance metric used in SLAs.</td>
</tr>
<tr>
<td>Term</td>
<td>Full Name</td>
<td>Explanation</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SLO</td>
<td>Service Level Objective</td>
<td>A specific range of acceptable values for an SLI (performance metric) used in SLAs.</td>
</tr>
<tr>
<td>TB</td>
<td>Terabyte</td>
<td>A storage unit equal to 1,024 gigabytes or $1,024^4$ bytes.</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transfer Communication Protocol/Internet Protocol</td>
<td>Suite of communication protocols to transmit information over a network (commonly, the Internet).</td>
</tr>
<tr>
<td>TPS</td>
<td>Transparent Page Sharing</td>
<td>An Operating Systems’ technique for sharing identical and read-only memory blocks among processes.</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
<td>A system’s piece of software that emulates an operating system.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Our society increasingly relies on computing services to solve various types of problems. We use computing systems for storing files, analyzing large data sets, performing predictive business analytics or just playing online games. All these services are made available to us through a number of interconnected devices and computers. The software that powers them up, however, is not running entirely on our local devices. Instead, parts of these applications run remotely on servers, interfacing with local clients and apps. The server-side parts of these applications are mostly written in high-level computer languages, which are usually interpreted and executed by language runtimes. Although this remote software can be run individually on its own private computer, cloud computing is usually preferred due to its easier maintenance, better scalability and reduced start-up costs.

Clouds abstract computing resources, provision them to tenant clients and
enable the resources’ transparent sharing among tenants. This ability to support multiple tenants sharing some service is called multitenancy. Cloud services are usually provided in a pay-per-use manner; hence, the suffix -as-a-Service is added to their description to denote this.

The provided services are described in formal agreements called Service Level Agreements (SLAs), which include performance targets the provider is expected to achieve; these targets are referred to as Service Level Objectives (SLOs). Cloud providers aim in satisfying SLOs mainly through scaling to adapt as the load fluctuates. This is crucial for the whole system to be operational; nevertheless, violating SLOs can also result in financial penalties.

The abstraction and sharing of cloud computing resources happens in various levels across the hardware/software stack: Infrastructure-as-a-Service (IaaS) concerns logistics, power, network, hardware and virtual machines; Platform-as-a-Service is about containers, frameworks, runtimes, services, application servers; and Software-as-a-Service provides user authentication, multiple organizations and users, personalized display and functionality.

Cloud computing is a very active research field and increasingly more systems rely on it [37]. As such, this thesis contributes in our understanding of how multitenant platform clouds perform and satisfy their SLOs, particularly in the presence of aggressive co-located tenant applications. The focus is placed on expanding multitenancy; modeling performance interference; improving scaling times; modeling and satisfying request-based SLOs; and exploring and mitigating the effects that dynamic memory management exerts on the
Figure 1.1: Flow of the research chapters: Chapter 3 proposes multitenancy at the application server. This came at a cost of reduced performance; thus, Chapter 4 investigates the effect of multitenancy interference in performance and scaling. A technique to mitigate performance interference and reduce startup times in cloud applications is presented in Chapter 5; a multitenant-load aware reordering technique is presented in Chapter 6. Garbage Collection effects in multitenancy became apparent; their impacts are studied in Chapter 7 and mitigated in Chapter 8.

tenants of a cloud (Figure 1.1).

More specifically, the background of this field is further elaborated in Chapter 2. Chapter 3 sets off by investigating a novel cloud multitenancy mode, that of Application Server as-a-Service (ASaaS). The main component of ASaaS is a multitenant application server, which enables tenant applications to securely run on the same process; therefore, maximizing resource sharing benefits. Such a multitenant application server is experimentally evaluated through a Java EE prototype built on top of IBM's multitenant JVM.

Chapter 4 investigates performance interference on containerized platform clouds and its effect on scaling. This is conducted with the help of a set of Cloud Burner tenants that target and consume specific resources. Additionally, using these Cloud Burners, a novel application classification system is
proposed that uses the slowdown a tenant experiences while a specific resource is over-consumed. A mathematical model of the scaling performance of tenants for clouds that follow the CPU-shares allocation strategy is then proposed and experimentally evaluated.

Chapter 5 tackles the problem of reduced performance and increased performance interference while scaling. To this end, the startup and warmup time of scaled and restarted instances is improved through sharing dynamically compiled artifacts from the first to the subsequent instances. Experimental evaluation of the Dynamically Compiled Artifact Sharing (DCAS) technique confirm its potential in improving the performance of the scaled tenant as well as minimizing the scaled tenant’s impact on its neighbors.

Chapter 6 discusses the problem of responding on-time according to SLAs and proposes a theoretical model that describes the expected on-time performance of a cloud system for a given number of clients. The model also results in a theoretical, constant time calculation of the ideal number of instances required for a certain load. The model is experimentally evaluated and its insights are then used to propose and evaluate a technique that increases the on-time satisfaction ratio through reordering the execution of requests depending on their type and the system’s load.

Chapter 7 explores the effects of Garbage Collection (GC), which is the process of automatic memory deallocation offered by language runtimes, on cloud applications. *CloudGC*, a versatile and configurable benchmark is designed to explicitly target this component of the runtime. Using *CloudGC*, the four
GC policies of IBM’s production Java runtime, the J9 JVM, are evaluated in terms of SLO satisfaction.

Chapter 8 proposes a technique for scaling components of an instance, while the load on this instance fluctuates. Idleness and/or low-load periods are utilized to reduce heap memory as well as restrict the resources GC tasks consume. This runtime scaling is mostly beneficial for co-located tenants who experience reduced interference, but it can also improve the performance of the scaled tenant since the down time is used to prepare it for the upcoming high loads. Finally, Chapter 9 concludes by presenting an overview of the conducted work as well as the future research opportunities and problems this thesis enables.
Chapter 2

Background

The interconnectivity offered by networks has enabled web and distributed applications over a plethora of devices (Figure 2.1). Personal computers, tablets and laptops as well as embedded systems, sensors and actuators communicate with each other, constituting the Internet of Things (IoT) [58]. The various IoT elements produce and receive a copious amount of information that is formatted in various ways and requires tight turnaround times [79]. A common practice is to execute the processing and storage of this type of big data operations on centralized nodes. These computational nodes are implemented by computer clusters, also known as data centers, that are distributed across the globe. Pay-as-you go clouds are used to abstract computing resources made available by these clusters as well as to unify remotely located subsystems through a single point of entry [113, 68]. Consequently, applications are running in a distributed way in this environ-
Figure 2.1: The contemporary Internet (of Things) connects multiple types of devices to each other; clouds frequently act as a centralized point of contact for communication, processing and storage purposes.

ment for multiple reasons: First, they are split into two parts, one that runs on the IoT edges (client-side) and one that runs on the centralized nodes (server-side). Second, due to the sheer number of edges, a single computer is not sufficient to serve all its clients without overloading. Instead, centralized services are distributed on one or multiple clusters or clouds. In addition, microservice oriented designs on the server-side break down its functionality into independent and specialized parts. Consequently, applications are additionally distributed across a number of functionally distinct components that rely on each other to complete a greater task [117].

To maintain isolation and simplify the deployment of applications, cloud providers virtualize their resources. Thus, even though an operating system still manages the provisioning of hardware resources on a node, virtualization managers harness the host operating system and expose parts of these resource to the users.
For both client- and server-side, high-level languages are predominantly used as they speed up development, are cross-platform and minimize the risk of introducing bugs and security exploits [78].

The details of this environment are further elaborated in this chapter.

2.1 Operating Systems

An Operating System (OS) is a special type of software that controls access to a computer’s physical resources and as such, crucial in understanding the performance of cloud systems (Figure 2.2). The OS cooperates with the hardware to provide a protected environment for user code. Interaction with the hardware happens only through an interface of system calls that the OS exposes to user applications [124].
OSs allow multiple user applications to securely run concurrently and/or in parallel. Processes are used to enforce resource isolation guarantees. Each process has one or more threads (active pieces of code) that execute—potentially on top of multicore hardware. Threads of the same process share the same virtual address space; the OS ensures that they cannot access private memory associated with other processes. Additionally, the OS regulates threads’ access to the CPU via scheduling. OS schedulers are controllable with various parameters—including priorities, ratios and limits—to allow the OS manager to regulate access to the CPU accordingly [124].

2.1.1 Linux Control Groups

The Linux OS offers control groups (cgroups) for managing process resources. A cgroup is a set of tasks (and their future children) with certain resource limitations, priorities, usage accounting and control. A cgroup can be forced/pinned to run on a subset of the available CPU cores, a subset of available memory nodes or be given a maximum amount of resident memory it can occupy. Additionally, a cgroup can be forced to run at most for a certain number of clock ticks per window of time. Furthermore, cgroups also support CPU shares, which guarantee a minimum CPU proportion [81].
2.1.2 Virtualization

OS-level virtualization abstracts the provisioning of OS-related resources. Users of a virtualized OS perceive it as if a whole machine were dedicated to them; however, they might actually be transparently sharing the hardware with other tenants. Two main types of virtualization are commonly used, Virtual Machines (VMs) and Containers [124].

VMs provide their users a full image of a guest OS that controls a set of available virtualized hardware resources. VMs are executed by hypervisors that are tasked with controlling and exposing physical resources to the guest OS. Two main types of hypervisors exist: Type-1 and Type-2 [124].

Type-1 hypervisors run directly on the host’s hardware; they spin up guest OS images to their tenants. System calls originating from type-1 VMs are routed through hypervisor hooks to the machine. Examples of Type-1 hypervisors are Xen\(^1\), Oracle VM Server\(^2\) and Microsoft Hyper-V\(^3\).

Type-2 hypervisors run on top of a host OS; they spin up VMs that are essentially processes of the host OS. System calls originating from a guest OS are sent to their Type-2 hypervisors that route them to system calls of the host OS. Examples of Type-2 hypervisors include VirtualBox\(^4\) and QEMU\(^5\).

Containers are colloquially referred to as lightweight virtualization. Unlike their VM counterparts, they do not provide a new OS instance to their tenant.

---

\(^1\)https://www.xenproject.org/
\(^2\)www.oracle.com/Oracle_VM
\(^3\)www.microsoft.com/hyper-v-server/
\(^4\)https://www.virtualbox.org
\(^5\)https://www.qemu.org
2.2 Cloud Computing

Cloud computing [68, 101, 47, 60] abstracts resources and provides them as-a-Service to customers (tenants) of a cloud provider in a pay-as-you-go manner. Multitenancy on the cloud enables the sharing of infrastructure, network, hardware, computing, software and other resources between tenants. Depending on the level of sharing, cloud providing systems can be grouped into the following three main categories [107, 126, 110, 28, 135] (Figure 2.3):

Figure 2.3: Cloud computing service models include IaaS, PaaS and SaaS. Each abstract and provide increasingly larger pieces of the hardware/software stack to their clients; they also tend to hierarchically leverage each other.

Instead, they provide an isolated view of the existing OS that includes unique process data, user data, networking and mounted file systems. They are commonly implemented via cgroups [29].
Figure 2.4: A sample architecture of a PaaS cloud system that manages a number of hosts and deploys user containers on them. PaaS containers commonly implement large parts of the software stack required by the user.

**Infrastructure-as-a-Service (IaaS)** concerns the sharing of physical space, logistics, cooling, network, hardware and physical computers through the use of VMs. Tenants have access to remotely located hosts which they manage. IaaS clouds include Amazon AWS\(^6\), Windows Azure\(^7\) and IBM Bluemix\(^8\).

**Platform-as-a-Service (PaaS)** clouds enable the transparent sharing of the hardware/software stack. Tenants have reduced control, in comparison to IaaS clouds, as they are required to abide by further restrictions set by the provider. PaaS clouds support a number slots that tenants place their code within. These slots already provide a large part of the software stack;
the cloud also contains a number of external modules and services that the
tenant (and its code) can utilize (Figure 2.4). PaaS clouds and software
include IBM Bluemix\(^9\), Google App Engine\(^10\), Red Hat OpenShift\(^11\).

**Software-as-a-Service (SaaS)** refers to the highest level of multitenancy
where the tenants share all aspects of the application including hardware and
virtually all of the software stack as well as any external services. Tenants
are distinguished by means of authentication or identification. Unlike the
other two categories, SaaS usually does not require development. Instead,
tenant-administrators are commonly setting up their system appropriately.
SaaS clouds include Salesforce.com\(^12\), Microsoft Office 365\(^13\) and Slack\(^14\).

### 2.2.1 Service Level Agreements

Service Level Agreements (SLAs) describe both the functional and the non-
functional requirements of the system. SLAs constitute a set of binding
contractual agreements between clients and providers, who agree on a set of
services, compensations and penalties. The expected performance and Qual-
itity of Service (QoS) of the service is outlined by Service Level Objectives
(SLOs), which describe certain thresholds the provider is expected to main-
tain on a variety of Service Level Indicators (SLIs) that measure the system’s

\(^9\)https://www.ibm.com/cloud-computing/bluemix/
\(^10\)https://cloud.google.com/appengine/
\(^11\)https://www.openshift.com/
\(^12\)Salesforce.com
\(^13\)https://www.office.com/
\(^14\)https://slack.com/is
performance [24]. For example, an SLI could be the 90th percentile of the response time of responses over an 1-minute window; an SLO for that SLI could be that this time cannot exceed one second.

2.2.2 Scaling

A common method to satisfy SLOs in cloud computing is through scaling. Modifying an instance’s resources is called vertical scaling (scale up) and modifying the number of instances is called horizontal scaling (scale out). For scaling out in particular, load balancers are responsible for spreading incoming requests to scaled instances. Scaling can be performed either manually or automatically [68, 113].

2.2.3 Cloud Foundry

Cloud Foundry is PaaS software written in Ruby [4] and is used by various PaaS providers, such as IBM Bluemix\(^{15}\), Pivotal\(^{16}\) and SAP Cloud Platform\(^{17}\). Cloud Foundry requires various types of VMs, each serving a particular function, which can be scaled out individually. Multiple Runner VMs, also known as Droplet Execution Agents (DEA), execute the code of the tenants. Each Runner VM is split into multiple Warden containers, which are implemented with Linux containers.

\(^{15}\)https://www.ibm.com/cloud-computing/bluemix/
\(^{16}\)https://pivotal.io/platform
\(^{17}\)https://cloudplatform.sap.com
Tenants can run their code using Buildpacks, which describe how to create the image of a running container (droplet). Thus, a Buildpack downloads and installs all the necessary applications required for the tenant’s code to run as well as the tenant code itself. For example, a Java Buildpack will install a Java Runtime and a Java application server to run the Java files provided by the tenant. Afterwards, the container can be connected with various external services through the Service Broker API. Both Cloud Foundry and IBM Bluemix support a number of Buildpacks with which tenants can run their Java, Python, Go, Node.js, Ruby or PHP code. The majority of the experimental results presented in the following chapters have been conducted on an isolated and local installation of Cloud Foundry.

2.2.4 Docker

Docker is an operating system virtualization software that enables the packaging of an application and its dependencies on a portable image that can then be deployed on a variety of hosts. A docker image is created through a docker script. Docker scripts can inherit from one another; thus code replication is minimized and reusability is promoted. After an image is built, it can then be uploaded to an image server, which makes it available for download after authorization. Docker images are deployed on containers; consequently, multiple Docker containers can be transparently deployed on the same host. Multiple hosts can also be used with the addition of an overlay virtual network and a load balancer [83].
Docker and its PaaS version, Docker Swarm were used in Section 5 for experimentally evaluating a proposed technique.

2.3 Language Runtimes

Cloud services are predominantly implemented in high-level languages such as Java, C# and Node.js. These languages are not directly compiled into machine code; instead, they are interpreted by a program called a language runtime, which executes it on the host machine [11].

State of the art language runtimes offer a plethora of performance and profiling services [10]. For example, a Just-in-Time (JIT) compiler is used whenever a piece of user code is executed frequently. The JIT is then invoked to compile frequently executed pieces of code to native machine code, which can then be executed directly leading to massive speedups [56]. Additionally, high-level languages prescribe and language runtimes implement dynamic memory management, which is commonly referred to as Garbage Collection (GC). Developers do not have to keep track of memory they no longer need and explicitly deallocate it; instead, the language runtime ensures that unused memory is reclaimed automatically [65].

Language runtimes include the IBM J9 JVM [10] and Oracle HotSpot [12] for Java, JSP, Jython and other JVM languages; Zend Server [103] for PHP; CPython [112] and PyPy [34] for Python; V8 for Node.js [127]; and Microsoft’s Common Language Runtime [102] for .NET languages.
### 2.3.1 Java and Java EE

Java [55] is a widely used programming language. Java is first compiled into machine-independent bytecode, which is then run by Java’s language runtime, a Java Virtual Machine [78] (JVM). Industrial JVMs include IBM’s J9 [10] and Oracle’s HotSpot [12].

The Java platform Enterprise Edition [43] (Java EE) is an API and runtime environment specialized to support Java applications running on web platforms—therefore, suitable for deployment on the cloud. Servlets are part of Java EE and are used as a means of providing stateless code execution serving HTTP requests. Java EE application servers implement these features and provide a platform for developers to run their Java code. Examples of these servers are IBM WebSphere Liberty [17] and Apache TomEE [2].

The majority of the techniques and applications that are presented in the following chapters of this thesis have been implemented in Java and particularly, Java EE.

### 2.3.2 Garbage Collection

High-level languages create and maintain an object graph while they execute. The roots of the graph are located in the various stacks maintained per thread as well as various static and global object pointers [55] [78]. The framework and the standard library classes used by the web application also utilize a number of interconnected objects and so does the code of the web application.
Figure 2.5: Language runtimes split their memory into stack and heap. Data on the stack is available to the program depending on the running function’s scope. Objects are allocated on the heap and are alive when they are accessible through a chain of links from the stack; otherwise, they are dead.

High-level languages offer explicit allocation of new objects but not their explicit deallocation. Instead, the runtime has available a certain slice of memory, referred to as the heap, to store the objects of the graph. Each object contains a number of slots that can point to other heap objects as well as a portion dedicated to storing non-pointer-data such as integers, floats and chars. The running application, which includes standard libraries, frameworks, application servers and the web application itself, is referred to as the mutator because it has the power to alter the stored graph [55] [78] [65]. An example of this setup is displayed in Figure 2.5.

The mutator can have multiple threads, each associated with its own execution stack, which stores the data its current method is working on as well as the data of any other methods that have invoked it recursively. These
thread-stacks, alongside some native constructs the runtime uses, store references to the root objects on the heap. When a mutator thread invokes a function, a new stack frame is created and a number of data, including some heap references are passed on to it. Afterwards, when the mutator returns from this function, the top stack frame is discarded and any references it contained to root objects are lost. Consequently, this pushing and popping of stack frames by the mutator threads modifies their root set by adding and removing object references respectively. Additionally, the mutator thread can also modify the object pointers of its top stack frame, which allows the further addition and removal of objects to the root set [55] [78] [65].

Also, a mutator thread can invoke a special runtime instruction to allocate a new object. Assuming that there is enough space on the heap for the time being, after a portion of the heap is reserved for this new object, its starting position is returned and stored on the thread’s stack (or a internal construct of the runtime in some special cases). Additionally, the mutator thread can also modify a reference of an object to point to another object or to null, which is a special pointer that indicates that no connection exists for this slot. Finally, apart from mutating the object graph, mutator threads perform other tasks such as reading and writing payload data on the objects, reading pointers from object slots as well as reading and writing data [55] [78] [65].

As described, the mutator thread has no ability to deallocate objects, which will eventually result in the heap memory being depleted. When this happens, GC threads are triggered to start the memory reclamation process.
The goal of the GC is to find and remove objects that are not reachable from the root sets; these unreachable objects are referred to as dead objects. The dead objects are also called garbage and thus, the term garbage collection was coined [55] [78] [65].

Additionally, because the mutator might be modifying the object graph if allowed to run concurrently with the GC, there is a risk of data races. This can invalidate the integrity of the memory and lead to a range of undesired behaviors: from incorrect output to a complete system crash! Thus, GCs utilize various barriers and locks to prevent mutator threads from conducting such operations. In the worst case, a stop-the-world phase might be triggered, which blocks all mutator threads (including of course the ones running for the cloud service) until the GC is finished [55] [78] [65].

### 2.4 Overview

This dissertation places itself in the intersection of Language Runtimes and Platform Clouds with a focus on modeling and improving the performance and SLOs of both provider and client. In particular, connecting the cloud with the underlying OS is crucial as it is the OS that essentially provides access to the hardware that executes cloud services and runtimes. In the next chapter, a multitenant application server for the cloud is discussed.
Chapter 3

Application Server as-a-Service

Applications that run on the cloud can be deployed more efficiently, if sharing of their software stack is increased to higher levels. In this chapter, a novel cloud model that securely deploys tenants on the same application server process is outlined. The main idea is to enable the provisioning of application server slices as-a-Service, while security and performance isolation guarantees are maintained. Elements of this work have been published in the Proceedings of CASCON 2015 [90].

3.1 Motivation

The standard way of sharing cloud resources requires extra memory because even though tenants can have the same application stack as high as their application-specific code, these common parts have to be replicated across
Figure 3.1: High-level memory sharing architecture of the ASaaS multitenant technique on an example enterprise stack.

all hosts and instances. In particular, cloud services tend to utilize various components called microservices. Microservices are commonly small enough that the added overhead of being deployed on separate servers might not compensate for their benefits. To alleviate this, a new type of cloud multitenancy is proposed that securely shares a larger part of the application server stack among tenants running on the same host (Figure 3.1). Thus, by reducing memory requirements, tenant applications can be packed more tightly on the cloud.

The design of this new cloud multitenancy model is discussed. Additionally its key component, a multitenant application server is designed, implemented and evaluated on top of IBM’s multitenant JVM. A theoretical model with the expected gains is also created and evaluated. Overall, this work paves the way towards a more efficient cloud provisioning model, that of Application Server as-a-Service (ASaaS).
3.2 Runtime Isolation

Java specifications and its class loaders provide partial security isolation between running tenant applications on the same JVM. In particular, Java does not allow direct memory manipulation or pointer arithmetic; therefore, one tenant cannot read or modify objects belonging to another tenant unless they explicitly get a reference to such an object.

Furthermore, the hierarchical way that Java class loaders work ensures that identically named classes of more than one tenants will not result in conflicts. To this end, classes are uniquely identified by both their name and their class loader; thus, as long as each tenant has its own loader for its class files, no ambiguity exists. Nevertheless, Java Class Library (JCL) files are loaded by the system class loader. Therefore, separation does not happen for these classes even with separate child loaders, a feature already provided in Enterprise ARchive (EAR) projects for instance.

However, to increase the level of sharing, tenants can share the loading of the common JCL files. This can be achieved by using one parent loader; nevertheless, a new set of problems arises by doing so. The class objects themselves, as well as any static fields they may contain, will be automatically shared amongst tenants. This immediately breaks the isolation guarantees required by multitenancy because one tenant may modify the value of a static field or even lock on its monitor or the monitor associated with the shared class object itself.
Apart from security isolation, traditional JVM implementations do not take into consideration performance and resource isolation. Consequently, if a tenant overuses a resource like memory or CPU, the performance of the other tenants is directly affected. Furthermore, if a tenant crashes, the whole JVM would also go down even if the remaining tenants were running normally. Examples of tenant-unsafe code are displayed in Listing 3.1.

```java
synchronized(String.class){
    // Unsafe because only one class monitor exists per JVM
}
System.out=null;
    // Unsafe because only one copy of this field exists per JVM
synchronized(System.err){
    // Unsafe because only one monitor exists per JVM for this field
}
```

Listing 3.1: Examples of tenant-unsafe code in standard Java.

3.3 Related work

Unlike ASaaS, IaaS and PaaS systems are not specialized in abstracting an application server to their clients; instead, they provide general computing resources and platforms. Additionally, IaaS and PaaS systems cannot increase the level of multitenancy to levels as high as ASaaS because of their more general isolation rules. On the other end, SaaS systems are too specialized
and normally exclude the ability to be generically programmed; instead, they provide a customizable but standardized interface. ASaaS, however, enables its clients to upload generic code targeting high-level languages, like Java EE. Similarly to SaaS, other as-a-Service models, such as Database as-a-Service or Storage as-a-Service, focus on providing highly specialized functionality to applications; thus, inherently lack the programmable flexibility of ASaaS. The ASaaS cloud model requires a runtime with isolation guarantees. The Java Specification Request (JSR) 121 concerns application isolation [13]. Even though JSR 121 includes a number of features that are required to be present for ASaaS, it has not become part of the standard Java specifications yet. Nevertheless, it has been used as a basis for isolation in a number of experimental JVMs.

Java runtimes that support tenant isolation include MVM [41], I-JVM [53] and KaffeOS [26]. These JVMs utilize a number of techniques to achieve isolation, such as modifying the Garbage Collector (GC) to keep memory of tenants separate, logging and limiting resources used by tenants, using intermediate pointers to enable tenants to share objects more safely, etc. In particular MVM and KaffeOS do not discuss the ability to manage tenants at the Java level and I-JVM provides an automated way of doing so in the OSGi [14] platform. Whereas, ASaaS requires a multitenancy API that will operate on a higher level and allow authorized Java users to manage tenants in a broader way.

Waratek [16] is a company that provides a multitenant JVM that uses virtual
containers to isolate tenants; however, it does so by restricting language support and in particular synchronizing on and modifying static fields. Elastica [6] is a product based on Apache Tomcat that uses Waratek’s multitenant JVM.

Watson [133] compared four different modes of the Liberty web server: standard, basic multitenant, sharing of the framework and sharing of both the framework and the OSGi bundles. The author’s results showed that significant memory savings were measured and also that the startup time of the applications was reduced when more sharing was enabled. More specifically, the author reports a memory footprint reduction of around 40% and startup time reduction of around 60% for framework and bundles sharing in the multitenant mode for 10 tenants. However, this thesis does not propose a modification that allows direct control of tenants at the Java level; it only discusses gains achieved by sharing class loaders and the Class Path (CP) on the IBM multitenant mode and the experiments only concern measurements during the startup time of the server without any client requests.

3.4 The Multitenant (RaaS) IBM JVM

An ASaaS model requires an underlying multitenant runtime (RaaS) for its multitenant server instances. The IBM JVM provides such a RaaS solution [64]. The IBM RaaS operates in two ways to enforce security and performance isolation among its tenants. First, it replicates all static fields and
monitors per tenant. Second, it enforces controllable utilization limitations in key resources, such as heap, CPU, number of threads and network bandwidth. This separation takes place through the per tenant TenantContext objects. However, a special, non-affiliated ROOT_TENANT context is also present that allows the JVM to securely manage tenants.

By starting up successive multitenant IBM JVMs with the -Xmt command line parameter only a single JVM process actually runs. Each JVM startup results in a new tenant being added to the first instance that runs as a daemon. The IBM Xmt mode has been implemented in the IBM JVM by an extension JCL package, which provides a number of multitenancy related features necessary to attach and start new tenants in the Java daemon. It is these methods of this package that are partially exposed to create the TenantAPI class discussed in Section 3.6.1.

Furthermore, when the Xmt mode is on, the user is also able to enable CP sharing amongst different tenants, which is only performed by an internal option. This allows tenants with the same class path to share classes that are loaded in the VM by a previous tenant using a common parent loader between tenants. Nevertheless, CP sharing is a general approach and it does not take into consideration any application specific sharable artifacts.

Finally, the IBM JVM also provides another configuration for achieving memory sharing and reducing memory footprint, which was used for the standard, one-JVM-per-tenant mode during evaluation. Class sharing [40] creates a cache where loaded classes are stored. Different JVMs can access it and ben-
efit by reducing the amount of memory needed, loading classes already in the cache faster and speeding up consecutive JVMs’ startup time.

Consequently, the IBM RaaS provides an alternative for running multiple application servers on the same runtime. For example, consider a docker file or buildpack that starts multiple instances of the Liberty server with the `-Xmx` flag. Such a mode (RaaS) is also considered and evaluated. Nevertheless, it still falls short on sharing all the portion of the stack that can be shared. Instead, the proposed ASaaS technique, apart from sharing the runtime also enables multiple tenants to share the same application server.

### 3.5 Memory Model

The memory requirements of a cloud client/server service can be broken down as follows: First, the runtime supporting the application maintains a number of internal data structures, including the compiled code of the tenant. Second, the application server classes running on top of the runtime maintain their own information, especially for abstracting the interface for TCP/IP communication to the upper level. Third, the tenant application itself, even if it is stateless, stores information, the very least regarding the request each of its threads is currently handling.

Even though various virtualization techniques could be used to enable memory sharing between co-located tenants, a unique portion of memory per process, server instance and server tenant will be present. Let these unique
footprints be $\text{mem}_{\text{Run}}, \text{mem}_{\text{Srv}}$ and $\text{mem}_{\text{Ten}}$ respectively and also let $n$ represent the number of tenants.

Additionally, since clouds operate in a distributed way, multiple instances are expected to be deployed on different hosts; consider $m$ as the number of available hosts and $E(i, j) \in \mathbb{N} \cup \{0\}$ the number of the $i$th tenant’s instances deployed on the $j$th host. Using this information, the memory requirements of three types of cloud deployments can be described.

In this study, three modes are considered: First, the standard mode refers to deploying multiple non-multitenant runtimes per host, each serving a different tenant through its own copy of an application server. Consequently the memory footprint needs to be replicated for each tenant:

$$
\text{Footprint}_{\text{Standard}} = \sum_{j=0}^{j<m} \left( \sum_{i=0}^{i<n} E(i, j) \cdot (\text{mem}_{\text{Run}} + \text{mem}_{\text{Srv}} + \text{mem}_{\text{Ten}}) \right) \quad (3.1)
$$

Second, the RaaS mode uses a multitenant runtime to deploy multiple tenants on the same host. Thus, only one runtime will be running per host but the application server cannot be shared yet:

$$
\text{Footprint}_{\text{RaaS}} = \sum_{j=0}^{j<m} \left( \text{mem}_{\text{Run}} + \sum_{i=0}^{i<n} E(i, j) \cdot (\text{mem}_{\text{Srv}} + \text{mem}_{\text{Ten}}) \right) \quad (3.2)
$$

Third, in the ASaaS mode, all instances of a host are running on a single runtime and application server, which eliminates the cost of keeping the
Figure 3.2: The memory models of the ASaaS mode versus the baseline standard and RaaS modes.

server data multiple times:

\[
\text{Footprint}_{\text{ASaaS}} = \sum_{j=0}^{j<m} \left( \text{mem}_{\text{Run}} + \text{mem}_{\text{Srv}} + \sum_{i=0}^{i<n} E(i,j) \cdot \text{mem}_{\text{Ten}} \right)
\] (3.3)

The overall organization of the different memory models is also displayed in Figure 3.2.

Consequently, the theoretical model predicts memory footprint reductions linear to the number of tenants for both RaaS and ASaaS modes in comparison to the standard. Furthermore, the model also predicts linear memory reductions of ASaaS over RaaS.

### 3.6 Implementation of a Java ASaaS

Implementing a proof-of-concept ASaaS system in Java requires a multi-tenant runtime. The multitenant IBM JVM was used for this purpose. How-
ever, the tenant management facilities it provides are internal and not immediately available to the Java user. Consequently, a new API for managing tenants is needed to expose these hidden functionalities, before a multitenant application server can be built on top of it.

### 3.6.1 TenantAPI

The multitenant IBM JVM provides a number of classes and functions at the Java level that are used to start tenants in the Java daemon process. This package was modified to provide a secure handle to these internal multitenant functionalities. A new JCL-like class, called TenantAPI, was designed, implemented and linked with the other JCL classes of the IBM JVM. In particular, the method signatures of TenantAPI are displayed in Listing 3.2.

```java
public static class TenantAPI{
    public static Object fullCreate(String id, String classPath)

    public static boolean run(Object tenantContext, Runnable code)

    public static boolean destroy(Object tenantContext)

    public static ClassLoader getCurrentClassLoader()

    public static String getCurrentTenantID()
}
```

Listing 3.2: TenantAPI methods.
To ensure secure access to the API, all TenantAPI functions include a check in the beginning that prevents non-root tenants from invoking them. The specifics of each function are as follows:

**Tenant creation:** Each new tenant is identified by a String id and a class path where its loader will be looking for classes. The new tenant loader created is a child of the loader of the ROOT_TENANT. The function returns a TenantContext object for the newly created tenant but it is first cast to Object so that the user cannot interfere with it directly since the definition of the TenantContext class is package private.

**Tenant execution:** Given an Object, cast from a TenantContext, and a Runnable, this function executes the given code under the given tenant context. It returns true if the switch to the tenant context and the execution of the code was successful. It is ensured that the given Object is instanceof TenantContext

**Tenant destruction:** Given an Object, cast from a TenantContext, and a Runnable, this function destroys the passed tenant. It is ensured that the given Object is instanceof TenantContext

**Acquire tenant’s loader and ID:** These functions return the loader and the ID associated with the invoking tenant.

### 3.6.2 An ASaaS Java Server

Utilizing the newly exposed multitenant functionalities through TenantAPI, a multitenant Java application server that supports a subset of the Java EE
specifications was implemented.

Initially, the multitenant server searches for tenant files in a predetermined *dropins* folder, loads them at startup, initiates the relevant *TenantContext* objects using *TenantAPI* and creates a mapping between their names and the loaded data (Listing 3.3).

Afterwards, when a request from a client is received, it is parsed and depending on the request address, the specified class from the specified tenant class file starts execution in that tenant’s context, again using the *TenantAPI* execution function (Listing 3.4).

```java
// For each jar file
for (File file : list) {
    // Create new tenant context
    tenant = TenantAPI.fullCreate(tenantName, classPath);
    // For each class file in the jar:
    // One tenant can have many classes which represent different services
    while (enm.hasMoreElements()) {
        [...]
        // Associate class name with tenant
        // context in a ConcurrentHashMap
        map.put(...);
    }
}
```

Listing 3.3: Tenant creation and class name association during initialization.

For comparison purposes, a standard one-JVM-per-tenant mode was also set up that does not use any *TenantAPI* functionalities, which was also used with
// Thread that handles a new client request:
// Parse tenant’s id from the request
// Request tenant context from map
tenant = map.get(tenantId);

// Switch to tenant mode
TenantAPI.run(tenant, new Runnable(){
    @Override
    public void run(){
        // Load the main method
        // from the requested class

        // And run it
        method.invoke(...);
    }
});

Listing 3.4: Switching to tenant mode in order to serve a client request.

the basic Xmt multitenant mode of the IBM JVM (RaaS mode). It should be stressed that none of these three designs require changes to the Java code of the tenants. Consequently, all three modes can directly support the same tenant class files.

3.7 Experimental Evaluation

Two types of tenant were used for the experimental evaluation: a Hello World and a Database-Intensive application both written on Java EE and thus, also containing an application server when deployed. A load driver was used to repeatedly fire 8 parallel requests to each tenant. The response time of the server was recorded and the proc/statm pseudofile was periodically read to gather statistics regarding the resident and shared resident memory of the
server process(es).

As far as memory is concerned, the total resident memory of the applications is actually lower than the one used in reality because of shared memory pages. Consequently, for both the standard and RaaS modes, which require more than one process to run\(^1\), the total average shared memory among all processes was subtracted and amortized. In particular, the following formulas were used:

\[
\text{Resident}_{\text{avg}} = \text{Avg}\{\text{proc/pid/statm resident samples}\}
\]

\[
\text{Shared}_{\text{avg}} = \text{Avg}\{\text{proc/pid/statm shared samples}\}
\]

\[
\text{Memory} = \sum \text{Resident}_{\text{avg}} - \sum \text{Shared}_{\text{avg}} + \frac{\sum \text{Shared}_{\text{avg}}}{\#\text{Processes}}
\]

In the case of the ASaaS mode, the same formulas were still applied but because it was using only a single process, it simplified to just counting the resident memory, not excluding any shared.

Shared classes were switched on only for the standard mode, being the only one that can benefit from this feature, having multiple JVM processes. In the other two cases, shared classes were switched off to minimize any overheads; instead Classpath sharing was used, which is enabled by default in the multitenant IBM JVM.

\(^1\)IBM’s RaaS mode creates a very small proxy process per tenant but all the actual work takes place inside the javad daemon process. Thus, the standard mode requires \(n\), the RaaS \(n + 1\) and the ASaaS only one.
Figure 3.3: Average response time speedup (top) and memory footprint (bottom) for Hello World tenants.

Each test was repeated 16 times for a varying number of tenants, all on the same hardware (1.8GHz 16 Core/32 Thread Xeon, 4x E7520, Nehalem-based, 64GB RAM, CentOS 6, 4096 Bytes page size). The Hello World and Database-Intensive results are displayed in Figure 3.3 and Figure 3.4 respectively. The error bars in the graphs represent the (relative) standard deviation of the samples around the mean.

The measurements show that in both tests, large memory savings with increasing tendencies when more tenants were used were attained. In particular, for 8 tenants the average memory for ASaaS was around 35% and 60% of the footprints of the standard and RaaS modes respectively.

As far as response time is concerned, ASaaS was generally slower than the
standard mode but either faster or at least as good as the RaaS mode. In particular, the Hello World tests show that the proposed technique is slower when fewer than 5 tenants are used but faster for more. This is a probable indication that because these tenants simply print a message and exit, they do not stress the system enough to justify the cost of the additional processes. However, when the tenants have to perform a more intensive task by communicating with a DB, the ASaaS approach is around 10% slower. Moreover, the RaaS mode was always slower but also saved memory in comparison to the standard. Nevertheless, the ASaaS technique still required less than half the memory RaaS needed making it drastically more efficient. Finally, these findings confirm the theoretical model predictions as the mem-
ory benefits are linear to the number of tenants between all different combinations as predicted. The measured footprint savings of the database-intensive tests is displayed in Figure 3.5.

3.8 Conclusions

Adding multitenancy at the application server level can be used for deploying applications on the cloud more efficiently. This can be applied in a novel cloud business-model where operators provide slices of a single application server to tenant-clients in a pay-per-use mode. Following the nomenclature of the field, the name Application Server as-a-Service (ASaaS) is a good descriptor for such a technology.

In this chapter, apart from a proof of concept implementation of such a server, a theoretical model for memory savings was also proposed and evaluated. In particular, three modes were compared, a standard one-JVM-per-
tenant mode, a default multitenant Runtime-as-a-Service (RaaS) mode and the main contribution of this chapter, the Application-Server-as-a-Service (ASaaS) mode. The ASaaS implementation enables the sharing of the application server data but still isolates its tenants by building upon an existing RaaS system and making its tenant management features public via our TenantAPI class.

The proposed theoretical memory model predicts linear memory savings to the number of tenants among the various levels of multitenancy, which was confirmed experimentally. In addition, the response time of ASaaS versus standard was reduced by 10% for the database intensive tenants but was up to 10% faster when 8 Hello World tenants were used. Furthermore, the ASaaS mode was also 10% faster than the RaaS for the best case. Therefore, ASaaS allows increasing the density of a cloud as it is able to pack more tenants per unit of cloud hardware.

As future work, the multitenant application prototype can be applied on an industry application server, such as Liberty Profile. Furthermore, another interesting path the TenantAPI technique can take is that of safe execution of remote functions, conceivably on a microservices-oriented architecture. Finally, the isolation guarantees of multitenant runtimes paired with the process-management-like API exposed by TenantAPI could lead towards providing further OS-like functionalities. This enables treating tenants as OS processes towards a single address space OS in Java.
Chapter 4

Scaling and Interference

Multitenancy comes with tradeoffs as it can cause resource contention. This chapter focuses on the problems of performance interference on containerized cloud systems. A mathematical performance model is proposed and evaluated; a set of resource-specific cloud burners is introduced to test co-location slowdowns based on resource-consumption patterns. Elements of this chapter have also been published in the proceedings of CASCON 2016 [99].

4.1 Motivation

Users of PaaS clouds presume that unlimited resources are present and available to them, especially during scaling out new instances. However, this is not the case; PaaS tenants tend to run on shared hosts whose performance is connected to both the underlying hardware limitations as well as the be-
behavior of other tenants. Regarding CPU resources in particular, a prevalent choice regarding tenant isolation is using a CPU shares algorithm that is often implemented through Linux cgroups and does not provide a constant rate of performance\footnote{http://man7.org/linux/man-pages/man7/cgroups.7.html}.

For instance, even though the PaaS software Cloud Foundry sets hard limits on resources like virtual memory and disk, it does not do the same with CPU utilization. Instead, it uses CPU shares, which guarantee a minimum CPU utilization for every tenant that requires it \cite{5}. OpenShift uses the same algorithm \cite{15}. Heroku, which is also using cgroups/containers for isolation, provides SLAs with one CPU share per 512MB of memory \cite{7}.

This can be a crucial problem, particularly in terms of SLO satisfaction: performance can become erratic even with constant loads when a co-located tenant creates sporadic resource contention peaks. Thus, it is crucial to model and predict the CPU contention during scaling, as well as classify cloud applications based on the slowdown their specific resource patterns incur to co-located tenants.

### 4.2 Related Work

Work in performance isolation started in the pre-cloud, virtualization era: a benchmark \cite{80} and a set of isolation experiments \cite{121} used various stressing applications targeting specific hardware resources. The benchmark was used
to compare performance isolation of three types of virtualization systems. However, these works did not focus on resource-intensiveness of tenant applications; instead, they measured the level of isolation each system offers. In addition, the different virtualization methods that were tested reported near-zero performance interference for CPU-intensive applications; nevertheless, in container-based clouds that use CPU shares this is not expected to be the case.

A related work on PaaS systems looked into various PaaS architectures and found that simply increasing virtualized resources without adding equivalent physical ones at the same time might not increase performance proportionally [25]. A theoretical analysis on the reasons that this is happening as well as extending the experimental setup is required, however. Also, based on their resource patterns, it is also expected that some types of applications should still be able to scale out well.

Being a more mature field of study, SaaS systems have been more thoroughly investigated on their tenants’ resource consumption, isolation and placement. In particular, methods and tools have been proposed to directly measure and monitor the resources each tenant uses. These analyses led to threshold-based triggering of scaling [51, 50]; predictive identification of CPU-intensive tenants with Kalman filters and subsequently limiting of their load to keep the machine in good operating status [131]; a framework for performance isolation that reserves and partitions resources while enforcing condition-based limits [59]; and a set of theoretical scalability metrics targeting SaaS
systems [128]. All these works are based on measuring resource consumption directly; however, this might be too intrusive, unavailable on some platforms or might not translate linearly to actual performance.

Related work on IaaS systems takes place at the virtualization level. Performance isolation by artificially delaying VMs, putting them in round-robin queues or blacklisting misbehaving ones, has been performed by directly measuring the resource consumption of these VMs [73]. Also, prediction of performance interference among VMs has been used with a Markov Chain model to dynamically reallocate VMs on different machines [39]. Performance prediction has been done with pairwise placement of VMs (and subsequent classification) on the same hypervisor [72, 62], as well as CPU utilization prediction using Kalman filters [67]. Additionally, automated scaling, which was performed with direct measurements of resource consumption and, similarly to the systems described before, a set of threshold-triggered scaling rules running on an intermediate layer [111, 129]. Moreover, the economic aspect of cloud scaling has been discussed and modeled in [48]: the authors found that it does not always make financial sense to scale even if some SLAs are violated. Finally, work has also been conducted in exploiting security vulnerabilities arising through performance interference [108].
4.3 Cloud Burners

To controllably stress hardware resources and investigate their effect on multitenancy, a set of cloud tenants, which are referred to as Cloud Burners, is introduced that create resource-specific load on a cloud system. Each burner targets a basic hardware resource, and through its execution patterns, tries to hijack it. This creates shortages in specific resources, which can then create starvation to other tenants that might be placed on the same VM. The resources we target are: CPU, cache, resident memory, disk I/O and network I/O.

Cloud Burners were implemented in Java EE, including a simple HTML GUI that controls the starting/stoppping of each resource burner as well as its startup options. The source code has been open sourced, including a sample network drain application implemented in C, and is available on GitHub [97].

4.3.1 Implementation Details

The logic of each burner is based on the underlying OS and hardware architecture design patterns [125]:

**CPU Burner:** A large prime number is repeatedly tested to decide whether it is indeed prime or not. This burner repeats a tight loop without creating any objects or performing any I/O; thus, it explicitly targets the CPU. The user can control the number of threads.

**Cache Burner:** A small buffer is created and a number of threads repeatedly
update the first position of a random cache-line and yield the CPU. The buffer is shared among the threads so that sharing—both false and true—is increased, further stressing the system’s cache. The user can control the number of threads, the shared buffer size and the line size.

**Resident Memory Burner:** Multiple large buffers are created for each thread, which repeatedly update the first position of a random page and yield the CPU. The buffers are not shared so that cache sharing is decreased and the main target of the burner becomes the resident memory. The user can control the number of threads, the buffers’ size and the block size.

**Disk I/O Burner:** Each thread uses a unique disk file and repeatedly opens it, writes a number in it and closes it. The reason that only this single memory reference is updated is to minimize interference with other resources. Regardless, the whole block needs to be written back on the disk every time a file is closed, explicitly targeting the disk’s I/O.

**Network I/O Burner:** A specific network server is selected and threads repeatedly send a message to it. The user can control the number of threads, the server’s name and port as well as the length of the message. A multiprocesssing server written in C is also implemented and provided, which accepts any incoming connections and simply consumes all the input without doing anything else with it, essentially acting as a network drain.
4.3.2 Experimental Analysis

Experimental results were collected to show the effectiveness of Cloud Burners in targeting their designated resources as well as investigating the effects of changing their default settings. Each Cloud Burner was run individually on an isolated and local installation of Cloud Foundry using their default configuration parameters. While each burner was running, the following metrics were recorded, each representing one of the main computing resources: CPU utilization, cache-misses, resident memory size, disk time and net throughput. These metrics were captured per second for a total execution time of two minutes on a local Cloud Foundry installation. The measuring took place using a Shell script that read information from various /proc/ files as well as profiling data gathered with perf.

4.3.2.1 Resource Targeting

The experimental procedure is the following: First, baseline measurements were taken without any burners switched on. Second, each of the burners were switched on one at a time after pushing again the application every time. The measurements were repeated three times for each case, using the default setting of the burners. The experimental results, displayed in Table 4.1, suggest that each of the burners successfully stressed its designated resource more than any other type of burner did. However, the other resources were also needed in various degrees, either because themselves or the cloud software environment required them.
Table 4.1: The experimental results of the resource consumption of Cloud Burners reveal that each burner stressed its intended resource more than its counterparts.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>All Off</td>
<td>22.04%</td>
<td>0.01</td>
<td>0.11</td>
<td>21.07%</td>
<td>0.00</td>
</tr>
<tr>
<td>CPU</td>
<td>99.97%</td>
<td>0.05</td>
<td>0.11</td>
<td>21.98%</td>
<td>0.00</td>
</tr>
<tr>
<td>Cache</td>
<td>99.65%</td>
<td>11.94</td>
<td>0.15</td>
<td>21.47%</td>
<td>0.00</td>
</tr>
<tr>
<td>Mem</td>
<td>42.25%</td>
<td>5.70</td>
<td>1.56</td>
<td>21.02%</td>
<td>0.00</td>
</tr>
<tr>
<td>Disk</td>
<td>32.45%</td>
<td>0.07</td>
<td>0.12</td>
<td>84.95%</td>
<td>0.00</td>
</tr>
<tr>
<td>Net</td>
<td>28.14%</td>
<td>0.13</td>
<td>0.12</td>
<td>22.19%</td>
<td>112.18</td>
</tr>
</tbody>
</table>

In particular, the CPU was moderately utilized by all burners and significantly utilized by the Cache Burner. This is expected because pressure in the cache means that load and store commands take longer to execute; in the meantime, the OS does not perform a context switch to the idle or some other process because there is no I/O interrupt. However, the threads of the Cache Burner yield on every iteration, which would allow tenants to run concurrently with it without starving for CPU.

All test modes registered cache misses; besides the Cache Burner, the Memory Burner also pressured significantly the cache but roughly half as much. This behavior is expected because there is no direct way of increasing the resident memory size without accessing new virtual addresses, which have to be first fetched in the cache and then to the CPU.

The measured resident memory was at most 150MB with the exception of the Memory Burner that required around 1.5GB. Any running application needs memory; large applications like a Java runtime with an application server running on top of it are expected to consume memory even without actively
processing requests. The disk was also stressed by the non-disk burners, a probable indication of our measuring tool’s activity, which was storing data in a file every second. Finally, the network was only used by the Network Burner; all other experiments measured zero bytes transferred.

All in all, Cloud Burners are indeed targeting and stressing the resources they were designed for.

4.3.2.2 Parameter Investigation

Various experiments were performed with different settings for each of the burners. All cases were repeated three times and each ran for a total of two minutes. For the CPU Burner, the number of threads was varied between 1 to 14 with the default setting being 9 since we ran this on an 8-CPU host. The results suggest that using fewer than 9 threads leads to lower CPU utilization; whereas using more, the already near 100% utilization remains the same. In particular, 32.16%, 62.12%, 99.97% and 99.98% average CPU utilization was measured for one, four, nine and fourteen threads respectively.
Figure 4.2: Number of cache misses caused by the cache burner and its resident memory set size for varying buffer sizes its threads used.

Figure 4.3: Number of cache misses caused by the memory burner and its resident memory set size for a varying number of parallel threads.

The Cache Burner caused the most cache-misses when 9 threads were used; there was a significant drop in cache-misses for the fourteen-threads test (Figure 4.1). This is an indication of contention among the burner’s threads and saturation of the system; if the threads are swapped too frequently, they do not have enough time to stress the cache as much. Also, increasing the buffer size of the Cache Burner causes more cache-misses but this comes at the expense of increasing the resident memory size (Figure 4.2). No significant differences in cache-misses when varying the line-size were measured.

Testing the parameters of the Memory Burner, no measurable differences
were found when the page size changed—the default being 4KB equal to the host OS’ page size. Additionally, higher buffer sizes resulted in higher resident memory sets—for example, setting it to 1,024MB caused average resident memory of 1,161.50MB. Setting its threads to a higher number had a greater impact to the cache without stressing the resident memory any further as shown in Figure 4.3; therefore, using one Memory Burner thread is sufficient.

Finally, the investigation for the disk and network burners revealed that these resources could be utilized to the maximum with two to four threads but without significant differences from using one or eight.

4.4 Resource Slowdown and Intensiveness

Cloud Burners can be used as a standardized way to measure performance interference among container-based cloud tenants. This in turn can result in a characterization of a cloud tenant by its intensiveness on a particular basic resource or, equivalently, in the reduction of its performance when that resource is highly contested. In both cases the application is profiled on a resource-slowdown-based analysis.

This unique characterization of a tenant-application can be acquired by switching on each burner one at a time and always keeping the same load. For each case, the application will slow down differently depending on how much it is affected by the lack of the resource the corresponding burner hi-
jacks. Thus, two application metrics based on this approach are proposed and defined in this section.

Consider the vector $B = \{\text{CPU, Cache, Mem, Disk, Net}\}$ of all Cloud Burners. Additionally consider the throughput $T$ of an application when it has the whole host for itself and $T_i$ when it shares the host with the $i$th Cloud Burner for $i \in B$. The Resource Slowdown Vector (RSV) of the application is defined as follows—notice that this definition can be easily modified in case other basic resources are deemed necessary:

$$\text{RSV} = \left( \frac{T_{\text{CPU}}}{T}, \frac{T_{\text{Cache}}}{T}, \frac{T_{\text{Mem}}}{T}, \frac{T_{\text{Disk}}}{T}, \frac{T_{\text{Net}}}{T} \right)$$  \hspace{1cm} (4.1)$$

For all burners, the throughput of the application can be at most as high as without the burners and in the worst case 0. Therefore, each of the individual slowdowns in an application’s RSV are between 0 and 1:

$$\text{RSV} \in [0, 1]^{|B|}$$  \hspace{1cm} (4.2)$$

RSVs contain higher numbers when their application performs better and lower numbers when worse. Next, the Resource Intensiveness Vector (RIV) of an application $A$ is defined, which operates in the opposite way—RIVs are also between 0 and 1 but this time higher numbers indicate higher demands on that particular resource:

$$\text{RIV} = \bar{I} - \text{RSV}$$  \hspace{1cm} (4.3)$$
Table 4.2: Description of the Java EE cloud-oriented applications implemented for testing purposes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld</td>
<td>A single JSP file that returns Hello World to the client.</td>
</tr>
<tr>
<td>100JSP</td>
<td>One hundred JSP files, all returning Hello World to the client.</td>
</tr>
<tr>
<td>Long Double Sums</td>
<td>Multiple threads per request, which calculate large sums and products of</td>
</tr>
<tr>
<td></td>
<td>integers and doubles including division and modulo operations.</td>
</tr>
<tr>
<td>Double Sum Array</td>
<td>For each request, a large array of doubles is allocated and traversed.</td>
</tr>
<tr>
<td>CloudTrader</td>
<td>A standard Java EE benchmark that performs stock-exchange transactions</td>
</tr>
<tr>
<td></td>
<td>with an external SQL database.</td>
</tr>
<tr>
<td>SemSleep</td>
<td>Each request acquires a Java Util Concurrent Semaphore, representing an</td>
</tr>
<tr>
<td></td>
<td>asynchronous external resource invocation, and holds it for 100ms before</td>
</tr>
<tr>
<td></td>
<td>returning.</td>
</tr>
</tbody>
</table>

4.4.1 Application Profiling

In order to demonstrate the proposed RSV and RIV metrics, a set of Java EE cloud applications were implemented and profiled. These applications are described in Table 4.2 and have also been open sourced on GitHub [98]. These testing applications were selected based on the variety of resources they are targeting. HelloWorld is expected to mostly stress the network because of its very high rate of request completion. Essentially, HelloWorld-like applications return back a result with a minimal amount of computation, which allows the next request to be sent; thus, a greater number of requests are possible to be satisfied that increase the network requirements of the system. Similarly, 100JSP should also be network-intensive, but because of the extra work required by the runtime to compile and maintain a hundred user classes, extra pressure is expected on the CPU, cache and memory. Long Double Sums is a CPU-intensive application but it should also stress the cache since it creates multiple threads. Double Sum Array is targeting the memory and indirectly, the GC of the runtime. SemSleep is representing
contention to external cloud resources and it should not be a bottleneck. Finally, CloudTrader [8], a standard Java EE benchmark, has varying and higher response times than the other applications and it is not expected that the network will be an issue; however, contention might arise at the disk level because it contains numerous Java classes and JSP files that need to be loaded and compiled by the runtime dynamically.

4.4.2 Experimental Evaluation

A local and isolated Cloud Foundry installation was used on a single hardware server and each of these applications were run on their own, one at a time. Apache JMeter\(^2\) on a different machine was used to create load with multiple parallel clients repeatedly firing requests to the application for 300s. This was repeated multiple times and the throughput of each application was calculated as the median throughput per second of the last 100s. This methodology was followed to exclude the startup and warmup time applications require until they ramp up to full capacity—the results indicated that performance normalized after around 150s for all applications. Each per-second throughput was calculated as the average across the multiple runs. Then, co-located Cloud Burners on the Cloud Foundry installation were switched one at a time. The same experimental process was repeated in order to calculate the applications’ throughput when each of the burners was on. Using this data, the RSVs of the profiled applications were calculated and

\(^2\)jmeter.apache.org
Figure 4.4: The experimentally acquired resource slowdown vectors of the tested applications are displayed with the use of radar charts. The lower the value on a specific resource, the slower the application became in the presence of the corresponding cloud burner.

are displayed in Figure 4.4.

The various profiled applications revealed varying resource patterns, which correlate with their design and implementation. In particular, HelloWorld and 100JSP are mostly network intensive, which is expected because both of them simply return a short message to the clients without any further processing. Consequently, due to the very large number of parallel requests they can support, a big part of the work actually happens in the frequent exchange of short network messages. Additionally, 100JSP is more intensive
in cache and memory than HelloWorld, which is also expected because it has a hundred extra classes that need to be compiled and maintained in memory. The LongDoubleSums application required mostly the cache and the CPU, which is also expected because it performs a long arithmetical calculation before returning the result. DoubleArraySum was also intensive with the network as well as with the cache and CPU because its replies to the clients require less time so it also produced a higher rate of network data. CloudTrader required mostly the CPU and the cache and secondarily the memory and the disk. Finally, SemSleep, was not intensive in any of the basic resources, which is expected because each request only acquires a semaphore and puts the thread handling it to sleep for some time; thus, no significant computing resources are required for SemSleep tasks.

4.5 Scaling Investigation

In this section, scaling in PaaS clouds is investigated focusing on interference and the CPU SLAs of Cloud Foundry as a basis. A mathematical model that describes how tenants share the CPU of a host is created. With the model, theoretical predictions are made and confirmed experimentally. Finally, tests are conducted on a commercial PaaS cloud that uses Cloud Foundry.
4.5.1 Theoretical Model

Each Cloud Foundry tenant is assigned a number of CPU shares, proportional to the virtual memory associated with it; the CPU shares are also capped between a minimum and maximum:

\[
\text{Shares} = \text{clamp}(a \times \text{Memory}, \text{Shares}_{\text{min}}, \text{Shares}_{\text{max}}) \quad (4.4)
\]

Each application is guaranteed to receive at least as much CPU as the proportion of its CPU shares on that host but only if it needs it. More formally, suppose a host that has been provisioned with \( n \) tenant instances, each with \( \text{Shares}_i \) CPU shares, for \( i \in \{1,...,n\} \). Thus, the CPU shares proportion of the \( k \)th instance is defined as follows and has to be more than 0, because of the minimum shares constraint, and at most 1 in the case of only one tenant on the host—all of them add up to 1:

\[
\text{Prop}_k = \frac{\text{Shares}_k}{\sum_{i=1}^{n} \text{Shares}_i}, 0 < \text{Prop}_k \leq 1 \quad (4.5)
\]

\[
\sum_{k=1}^{n} \text{Prop}_k = 1 \quad (4.6)
\]

Furthermore, from a tenant’s perspective, its CPU proportion on the host is calculated as follows, where \( A \) is the sum of the shares of all other tenants.
on the host:

\[
\text{Prop}_k = \frac{\text{Shares}_k}{A_k + \text{Shares}_k}, \text{ with } A_k = \sum_{i \neq k} \text{Shares}_i \quad (4.7)
\]

Consequently, linear increases to the number of shares a tenant holds, through linear increases of its memory, lead to sublinear increases in their CPU proportion. For the extreme cases, if a tenant is placed alone on a host or it has the vast majority of the shares, then its proportion is always 1; however, if \(A_k\) is much larger than \(\text{Shares}_k\), then the proportion becomes near linear to the number of shares:

\[
\text{Prop}_k = 1, \text{ if } n = 1 \quad (4.8)
\]

\[
\text{Prop}_k \approx 1, \text{ if } A_k \approx 0 \quad (4.9)
\]

\[
\text{Prop}_k \approx \frac{\text{Shares}_k}{A_k}, \text{ if } A_k \gg \text{Shares}_k \quad (4.10)
\]

Suppose that in a given time-window, an application \(k\) requires \(\text{Need}_k\) percentage of the CPU in order to satisfy all of its load. However, because other tenants are also running on the host, it will receive \(\text{Get}_k\). The lower bound of how much it actually acquires is decided by its CPU share proportion, enforcing the CPU SLA, and the upper bound, by how much it needs:

\[
\min(\text{Prop}_k, \text{Need}_k) \leq \text{Get}_k \leq \text{Need}_k \quad (4.11)
\]
Next, assuming that the OS shares any excess CPU evenly among any tenants whose needs have not been fully met yet, a bonus CPU per tenant can be recursively calculated as follows: First, the remaining CPU is divided by the total number of instances belonging to tenants whose needs have not been met. Second, any excess CPU tenants do not need is “put back in the pot”. Third, the process is repeated until either all tenants’ needs have been met or there are no more CPU resources remaining. The following recursive function calculates the expected CPU utilization each tenant $k \in \{1, ..., n\}$ is expected to receive in the $i$th calculation round:

$$
\begin{align*}
CPU_0(k) &= \min(Prop_k, Need_k) \\
CPU_i(k) &= \min \left( \frac{CPU_{Available}(i-1)}{More(i-1)}, Need_k - CPU_{i-1}(k) \right)
\end{align*}
$$

(4.12)

More($i$) counts the number of tenants that still need CPU after the $i$th round and $CPU_{Available}(i)$ represents how much CPU is still available to be allocated:

$$
More(i) = \left| \left\{ j \in \{1, ..., n\} : \sum_{m=0}^{i} CPU_m(j) < Need_j \right\} \right|
$$

(4.13)

$$
CPU_{Available}(i) = 1 - \sum_{j=0}^{i} \sum_{k=1}^{n} CPU_j(k)
$$

(4.14)

Finally, $\text{Get}(k) = \sum_{i=0}^{\max - 1} CPU_i(k)$ where $\max \geq 1$ is defined as the smallest integer such that $CPU_{max}(k) = 0$ for all $k$, which is the point that the recursive function stops evolving because all of the CPU has been allocated.
or the needs of all tenants have been met.

4.5.1.1 Model Predictions

A first outcome of this model is that, if only one tenant uses the host because every other tenants’ needs are near zero, then this tenant receives all of the CPU of the host regardless of the memory, shares and proportions of the tenants. Therefore, adding extra memory to it, by means of scaling out or up for example, will not give it any extra CPU. Also, this can be inductively expanded for any number of hosts a tenant’s instances are spread across and are still utilized only by this tenant:

**Theorem 4.5.1.** \( \text{Get}(k) \) is constant and equal to \( \min(1, \text{Need}_k) \) for any \( \text{Prop}_k \), if \( \text{Need}_j \) is 0 for all \( j \neq k \).

*Proof.* At any iteration, all tenants except for tenant \( k \) receive no CPU since they do not require any:

\[
(\forall j \neq k)(\forall i)(\text{Need}_j = 0 \Rightarrow \text{CPU}_i(j) = 0) \quad (4.15)
\]

Tenant \( k \) starts by obtaining the minimum between its needs and its CPU proportion:

\[
\text{CPU}_0(k) = \min(\text{Prop}_k, \text{Need}_k) \quad (4.16)
\]

At the next iteration, since there is no competition, tenant \( k \) obtains as much
CPU as it still needs:

\[
CPU_1(k) = \min(1 - CPU_0(k), \text{Need}_k - CPU_0(k))
\]  

(4.17)

And finally, either all needs have been covered or all the CPU has been divided thus the recursion ends at max = 2:

\[
(\forall j)(CPU_2(j) = 0)
\]  

(4.18)

Therefore:

\[
\text{Get}(k) = CPU_0(k) + CPU_1(k)
\]

\[
= CPU_0(k) + \min(1 - CPU_0(k), \text{Need}_k - CPU_0(k))
\]  

(4.19)

\[
= \min(1, \text{Need}_k)
\]

\[\square\]

Consequently, because CPU is a basic resource that applications must have in order to progress, it is predicted that a tenant will gain no speedups when scaling up their CPU on hosts that are utilized only by that one tenant. For scaling out however, speedups might occur because bottlenecks—locks, semaphores, thread pools, connection pools, GC activity, etc.—existing in a single instance can be divided across multiple instances as described by Amdahl’s Law.

A second outcome concerns how greedy, CPU-intensive tenants compete on
a single host. In particular, consider the first \( m \) tenants always need all of the CPU; whereas, the remaining \( n - m \) never need it. In this case, each tenant will receive its baseline CPU plus some extra, which is proportional to the sum of all proportions of idling tenants and inversely proportional to the number of greedy tenants:

**Theorem 4.5.2.** If \( m \in \{2, ..., n\} \) exists such that

\[
Need_j = \begin{cases} 
1 & \text{if } 1 \leq j \leq m, \text{ then} \\
0 & \text{otherwise} 
\end{cases}
\]

\[
Get_j = \begin{cases} 
Prop_j + \frac{\sum_{i=m+1}^{n} Prop_i}{m} & \text{if } 1 \leq j \leq m \\
0 & \text{otherwise} 
\end{cases}
\]

**Proof.** At any iteration, all tenants between \( m + 1 \) and \( n \) receive no CPU since they need none:

\[
(\forall j \in [m + 1, ..., n]) (\forall i)(Need_j = 0 \Rightarrow CPU_i(j) = 0) \quad (4.20)
\]

Tenants 1 to \( m \) start by obtaining their CPU proportion since their needs are greater than this minimum:

\[
(\forall j \in [1, ..., m]) (CPU_0(j) = min(Prop_j, Need_j) = Prop_j) \quad (4.21)
\]
And continue by splitting any remaining CPU between them:

\[
(\forall j \in [1, ..., m])(CPU_1(j) = \min\left(\frac{\sum_{j=1}^{m} (1 - CPU_0(j))}{m}, Needs_j - CPU_0(j)\right))
\]

\[
= \frac{\sum_{j=1}^{m} (1 - CPU_0(j))}{m} = \frac{\sum_{j=1}^{m} 1 - Prop_j}{m} = \frac{\sum_{j=m+1}^{n} Prop_j}{m} \tag{4.22}
\]

And finally, all the CPU has been divided thus the recursion ends at max = 2:

\[
(\forall j)(CPU_2(j) = 0) \tag{4.23}
\]

Therefore:

\[
(\forall j)(Get(j) = CPU_0(j) + CPU_1(j) \Rightarrow Get(j) = \begin{cases} 
Prop_j + \frac{\sum_{i=m+1}^{n} Prop_i}{m} & \text{if } 1 \leq j \leq m \\
0 & \text{otherwise}
\end{cases} \tag{4.24}
\]

However, tenant administrators have direct control only on the amount of memory they can request, which is then translated to their shares and subsequently CPU proportion as already discussed. Therefore, supposing \( B \) is equal to \( \sum_{i=m+1}^{n} Shares_i \), which is the sum of all proportions of idling tenants:
Corollary 4.5.2.1.

\[(\forall j \in [1, \ldots, m])(Get(j) = \frac{\text{Shares}_j}{A_j + \text{Shares}_j} + \frac{B}{m \cdot (A_j + \text{Shares}_j)}) \quad (4.25)\]

In case all tenants of the host are greedy, \(B\) is 0 and all tenants are allocated CPU equal to their proportion. This can be directly derived from Corollary 4.5.2.1 using Equation 4.5:

Corollary 4.5.2.2.

\[(\forall j)(\text{Need}_j = 1) \Rightarrow (\forall j)(Get(j) = \frac{\text{Shares}_j}{A_j + \text{Shares}_j} = \text{Prop}_j) \quad (4.26)\]

Finally, the observation of Equation 4.9 concerning a small ratio of shares on a host can now be extended to the CPU a tenant receives as follows:

\[\text{Get}(j) \approx \frac{\text{Shares}_j}{A_j} + \frac{B}{m \cdot A_j}, \text{if } A_j \gg \text{Shares}_j \quad (4.27)\]

And, if all tenants are greedy:

\[\text{Get}(j) \approx \frac{\text{Shares}_j}{A_j}, \text{if } A_j \gg \text{Shares}_j \text{ and } B = 0 \quad (4.28)\]

Consequently, it is predicted that linear speedups can only be achieved if a tenant has a very small share on the CPU in comparison to all other shares the remaining tenants hold on that host. Otherwise, the speedups are expected to be sublinear from Corollaries 4.5.2.1 and 4.5.2.2. Nevertheless,
other bottlenecks might prevent scaling even if more CPU is allocated linearly.

### 4.5.2 Scaling Experiments

To test these scaling predictions, a number of scaling up and out experiments were conducted on both a local Cloud Foundry installation on a single Linux machine as well as a PaaS cloud that uses Cloud Foundry. Six cloud applications were used as tenants, which are described in Table 4.2, each running alone and with a single instance on the system. For scale-outs, a second instance of each application was created; whereas for scale-ups, the resources of the existing instance were doubled. Apache JMeter running on a different machine was used for firing continuous requests with multiple threads on the deployed tenants for 300s times the total number of instances used on each test. The throughput of each case was measured and the median of the last 100s before scaling or ending was aggregated. All tests were run multiple times.

#### 4.5.2.1 No Contention Tests

Theorem 4.5.1 predicts that tenants running on a single host without competition will gain no speedup by scaling up. Instead, scaling out can be beneficial according to Amdahl’s Law, as long as internal bottlenecks that can be split across multiple instances are present.

In the local scaling experiments—without any of our burners switched on—
scaling up did not produce any measurable benefits as was predicted by the model. Furthermore, scaling out produced the expected speedup for the SemSleep application, which by design has an internal bottleneck in the form of a semaphore acquisition. CloudTrader also falls into this category since its performance only increased with scale outs—it is hypothesized that some internal limitation, like a connection pool size, is causing this effect. Additionally, scaling out the two network-intensive applications actually decreased their performance, apparently due to contention on an already saturated resource. Finally, the combined out and up scaling also did not produce any benefits for the applications without internal limitations. These results are displayed in Figure 4.5.

Figure 4.5: Experimental scaling results on a local and isolated Cloud Foundry installation.
4.5.2.2 CPU-Intensive Contention Tests

Corollary 4.5.2.1 predicts that multiple CPU-intensive tenants on a single host will receive CPU proportional to their shares and also share any CPU not used by idling tenants.

Consider a scenario where an idling tenant with 2GB of memory is placed on a host. Also, on the same host a CPU-intensive tenant, which is monitored, is placed with 0.5GB of memory and is subsequently scaled up or out; every scaling event will increase the total memory of the tenant by 0.5GB. Furthermore, a number of other CPU-intensive burner-tenants are also placed on that host, each with 0.5GB of memory.

Applying these numbers on Corollary 4.5.2.1, for 0, 1, 2 and 3 scaling events as well as 0, 1, 2 and 6 burners, the predicted CPU assigned to the monitored tenant is calculated and displayed in Figure 4.6. As already discussed, the prediction suggests that the tenant’s allocated CPU does not increase when scaling on an uncontested environment. However, as more and more burners are added, the increments of the allocated CPU become more and more linear. Also, the results are as such while the linear and not the clamping part of the memory to shares function is used: Cloud Foundry sets the minimum clamp to 8MB and the maximum to 8GB.

To test these predictions, the RSVs (Figure 4.4) of the profiled applications were examined. The two applications with the highest CPU and cache intensiveness were selected: LongDoubleSums and DoubleArraysSum. Then a scaling experiment with the same settings as the ones in the theoretical sce-
Figure 4.6: Theoretically predicted CPU allocation using the proposed model.

The scenario described above was conducted, using multiple Cloud Burner tenants running only in the CPU mode for the burner tenants. The throughput of these two applications was measured in all the described configurations and the aggregated results of multiple runs are displayed in Figures 4.7 and 4.8, using the standard deviation of the samples for their error bars (some are less than 2% and not visible on the graphs).

The following observations can then be made: First, the maximum predicted CPU allocation matches the experimentally measured throughput, especially for the scale out case. Also, the zero burners experiments produced no significant speedups for the scale up case. These measurements confirm the mathematical model.

Second, scale out performed better than scaling up; this measurement is consistent with Amdahl’s law, particularly because internal bottlenecks, such as GC, are present.
Figure 4.7: Experimentally measured throughput for the LongDoubleSums Java EE application, which resembles well the theoretical CPU utilization predictions of the model.

Third, the throughput of LongDoubleSums matched more closely the theoretical CPU maximum than DoubleArraysSum. Going back to the RSVs of the two applications in Figure 4.4, besides high CPU and cache intensiveness, DoubleArraysSum is also highly intensive on the network; therefore, it is not as aggressive with the CPU and could not outcompete the highly CPU-intensive burner. However, LongDoubleSums is mostly intensive in CPU and cache, whose share is definitely increased with scaling, and is also secondarily
Figure 4.8: Experimentally measured throughput for the DoubleArraySum Java EE application, which resembles the theoretical CPU utilization predictions of the model.

intensive on resident memory, another resource that becomes more available by scaling. This enables LongDoubleSums to utilize virtually all the extra CPU towards throughput.

Figure 4.9 displays the speedups after a single scaling experiment for all testing applications, with a single CPU-burner running alongside. Unlike the results of Figure 4.5, this time scaling produced speedups for LongDoubleSums and DoubleArraysSum. This is the case because these two applica-
Figure 4.9: Experimental speedup results of scaling the testing applications on the local and isolated Cloud Foundry installation with the CPU burner activated.

applications are primarily CPU- and cache-intensive as already seen in Figure 4.4. Nevertheless, HelloWorld and 100JSP still failed to produce any significant speedups. Again, inspecting Figure 4.4 suggests that these two applications are primarily network intensive. Therefore, because their bottleneck is not the CPU, the cache or the memory, which are the resources mostly increased during scaling on a single host, no extra network I/O was given to them and no significant speedups were obtained.

4.5.2.3 Commercial Cloud Tests

Besides the local isolated tests on a single host, scaling tests were conducted on a commercial cloud that uses Cloud Foundry. The scaling results, displayed in Figure 4.10 show similar patterns to those found in the local tests without contention. More specifically, scaling up did not produce any significant speedups; whereas, scaling out was able to increase throughput for
Figure 4.10: Experimentally measured throughput scaling speedups on a commercial cloud that runs Cloud Foundry (top 32 clients, bottom 64 clients).

the SemSleep application, which is again expected by design, and also for the LongDoubleSums, a probable indication that a new scaled out instance was placed on a different host, since scaling up for this application did not measure any speedups. Also, HelloWorld again had a small slowdown when scaled-out; as before, this has to be attributed to the network contention between the two instances.

CloudTrader runs have been especially variant. This is not the case in the local experiments, and the only difference of CloudTrader with the other applications is that it requires a database. Therefore, these discrepancies
should be attributed to the contention that must exist on the multitenant—low-end and free—database CloudTrader used on the commercial cloud. This contention can be the result of either the database itself having to handle multiple requests or the result of performance interference from other tenants on the same host (VM, container or hardware). Instead, for the local experiments, the database never had any contention and CloudTrader was able to make progress unimpeded.

To further investigate this, a long-term experiment was conducted in the order of multiple hours with 64 JMeter clients. The relative standard deviation of the per-second throughput of CloudTrader was 148.21%; whereas for DoubleArraySum, 21.14%. Therefore, this provides evidence that the measured performance variability of CloudTrader happens on the server-side and it is not related to network noise on the client, further supporting the idea that it is actually caused by database contention.

### 4.6 Conclusions

Multitenancy on PaaS clouds creates a set of questions regarding performance interference and scaling. Cloud Burners, a set of cloud tenants that explicitly target basic hardware resources, were introduced and their effectiveness in consuming and stressing their designated resources was measured. Second, using Cloud Burners, a set of resource-slowdown and resource-intensiveness metrics were proposed, which help profile cloud applications based on their
internal resource utilization patterns, regardless of their load. Finally, a theoretical model of CPU allocation and scaling on PaaS systems for interfering tenants placed on the same host was proposed and evaluated.

All in all, scaling does not always lead to improved performance, especially when this takes place on the same host and the tenant is already utilizing it as much as possible, or when the CPU is not the bottlenecking resource. Even worse, scaling out might lead to slowdowns due to contention among the instances for non-scaled-out resources such as the network. However, scaling out a CPU-intensive tenant across multiple hosts will interfere with the performance of other tenants on these hosts. Furthermore, applications with internal bottlenecks are improving their performance only when scaled out but not up. Moreover, tenants placed on the same host can have a significant performance impact on each other; especially when they compete for the same basic resources. Finally, the proposed RSV and RIV metrics (visualized in the radar charts of Figure 4.4) have added intuitive value in understanding these limitations.

Potential future work includes formalization of the options of the Cloud Burners per different architectures and also, new burners that target other basic resources, including a model of the slowdown caused to applications due to contention on external services, such as databases. Furthermore, more applications can be profiled with this technique and then, grouped and organized according to their resource intensiveness, which can lead to a novel performance-oriented classification method. Also, further research is required
to test how these findings apply to other related cloud technologies, such as Docker containers. Finally, the proposed methodology can be used for resource-slowdown-aware tenant placement on hosts.
Chapter 5

Dynamically Compiled Artifact Sharing

Clouds use scaling to elastically respond to load fluctuations and enforce SLOs. However, experience gained by the first instance cannot be easily shared with subsequent scaled instances. This leads to decreased startup performance and potential SLO violations for both the scaling and any co-located tenants, due to increased performance interference. This chapter outlines a technique for securely sharing dynamically compiled artifacts from the first to subsequent scaled instances of PaaS cloud tenants. Experimenting with the proposed technique resulted in significant improvements for both the scaled and any co-located tenants. Elements of this chapter have been published in the Proceedings of IEEE CLUSTER 2017 [88] and resulted in a patent application [42] with IBM.
5.1 Motivation

Platform clouds utilize scaling to satisfy SLOs as the load increases. New instances can be added or the existing ones can be restarted such that they acquire more resources. This can be performed both manually and automatically; in any case, application instances are isolated from each other and can be deployed on different hardware hosts and/or datacenters.

High-level languages commonly used on the cloud produce a number of dynamically compiled artifacts on startup and while warming up. For example, as the user code is interpreted by the language runtime, any classes it contains are converted to internal representations. Additionally, the JIT eventually kicks-in and converts any frequently used methods to machine code. All these are rather expensive tasks; if they are executing during the warmup and startup phase of every scaled instance, they can cause significant performance degradation, which can result in SLO violations or require additional cloud resources to overcome them. Furthermore, as it was previously established, performance interference will also increase for any co-located instances on the same host, which can inadvertently affect the performance of other tenants.

Solutions already exist on runtimes to mitigate these problems on a single host: Experience and artifacts collected by a first run can be shared with subsequent processes via memory-mapped files. Nevertheless, this approach does not work for clouds due to the strong isolation guarantees among con-
tainers and/or VMs. Additionally, scaled out instances are likely to be run on a distant host, cluster or even datacenter. Consequently, a cloud-oriented method is required to efficiently and securely share experience and dynamically compiled artifacts among scaled instances.

5.2 Dynamically Compiled Artifacts

The execution of high-level languages produces a number of data structures and native code depending on the user code and its execution patterns. To avoid repeating the same tasks, language runtimes have adopted techniques that enable the sharing of any immutable and dynamically created data across processes on the same container—or OS, if no isolation is enforced. For instance, when a Java bytecode class is loaded by a JVM, it splits its data to mutable (static fields, static locks, etc.) and immutable (bytecode, names and types of fields, and names and types of methods, etc.). The mutable data are private and specific to their owing instance. However, the immutable data are the same for any other Java runtime that uses the same class. Consequently, they can be safely shared with other Java runtimes that run on the same container to minimize resource consumption. In particular, the IBM JVM creates a Shared Class Cache (SCC), which contains all this immutable class data. The SCC is a memory-mapped file; consequently, the first process that creates it as well as any others that connect to it view it as part of their own address space and can directly reference any data on it [11].
The Java SCC is also used for storing Ahead of Time (AOT) compilation data of natively compiled hot methods and also, it stores JIT profiling data that provide performance hints [56]. Furthermore, it can also store key-value pairs that can be efficiently shared among JVM instances [52].

Further utilizing the Java SCC for improving performance has also been researched: Richard et al. found that a large number of String objects are used in Java applications [106]. Because Java Strings are immutable, the authors utilized the SCC to store and subsequently share Strings across instances of the same application on the same OS. Bierbrauer further expanded this technique to enable SCC-based sharing of any type of immutable object or array using a Java API [31]. In addition, Bhattacharya proposed using the SCC to store profiling data that enabled the fine-tuning of various internal structures of the runtime, which reduced memory footprint and improved startup time [30]. Such profiling data can include: program-counter-based escape analysis [63], object layout optimization [45, 44, 46] and Java Util Concurrent thread contention [89].

Apart from the SCC, Java EE has an option to precompile Java Servlet Pages (JSP) class files [43]. Additionally, the OPCache\(^1\) module in PHP stores compiled code in shared memory to avoid the cost of repeating the task on each request. Furthermore, the Native Image Generator\(^2\) of .NET creates local files containing platform-specific dynamically compiled machine-code.

\(^2\)https://msdn.microsoft.com/en-us/library/6t9t5wcf(v=vs.110).aspx
5.3 Related Work

One of the main benefits of the ASaaS technique, which was presented in Chapter 3, is that it shares dynamically compiled artifacts among tenants running on the same runtime [90]. However, sharing acquired experience can only happen when scaling takes place on the same multitenant JVM and not on a distributed cloud environment.

Cloneable JVM [69] can also share dynamically compiled artifacts but with a significant overhead and increased security risks. In particular, Cloneable-JVM stores the whole image of a running runtime, which can be subsequently copied and continued on a different host. Apart from the very large file required, the JVM has to be stopped on a non-critical point. Another risk is the loss of random pointer allocation: new instances of the same application can allocate the same data in different memory addresses. This is done to prevent security exploits and a Cloneable JVM-based solution would lack this defense mechanism.

Ahead of Time (AOT) compilation is a technique available in language runtimes. AOT compiles intermediate code, such as bytecode, to machine code. Unlike the JIT, it does so before the execution of the application. If AOT artifacts are present, the JIT does not need to construct them during runtime and the performance of the application might not suffer as much. Nevertheless, AOT does not utilize dynamic data and might create machine code for cold or unused methods.
Ogata and Tamiya [87] proposed utilizing Transparent Page Sharing (TPS) in OS hypervisors in order to increase sharing of Java class caches. TPS, supported by some hypervisors, is the detection of identical OS read-only pages on the same virtual addresses across a number of VMs and subsequently joining all of them in one physical copy. The authors’ solution was to modify the IBM JVM to always store the same classes in the same virtual addresses inside a shared class file repository. Therefore, when the JVMs finish updating their repositories and the TPS mechanism kicks in, the common pages are shared across a number of VMs on the same hypervisor. Bozek et al [36] also invented a similar TPS-based technique for hypervisors. Although these works share compiled artifacts, they do so after they are created and the TPS mechanism discovers them, which means that benefits are more over the execution lifetime rather than when the scaling takes place. Furthermore, these techniques work only for sharing artifacts on the same hypervisor. However, PaaS systems do not guarantee that instances of the same application will run on the same hypervisor, and different applications are less likely to be using Java or even loading the same classes. Moreover, not all hypervisors support TPS; therefore, this implementation is dependent on such a feature.

5.4 Dynamically Compiled Artifact Sharing

Dynamically Compiled Artifact Sharing (DCAS), which is the proposed solution to the aforementioned problems, shares dynamically compiled artifacts
between different instances of the same cloud application during scaling or application restarts. The technique focuses on the mechanisms of Cloud Foundry but can be extended in similar PaaS Clouds.

Since dynamically compiled artifacts were not designed to be shared on the cloud but on the same container, the main novelty of the technique is enabling the secure sharing of artifacts over the cloud.

DCAS relies on socket communication, which does not violate cloud isolation guarantees. An artifact server, also referred to as the DCAS Service, is used to first store and subsequently retrieve dynamically compiled artifacts. The DCAS Service securely stores per-tenant artifacts; privacy is guaranteed via unique identifiers provided by the cloud manager as well as a secret key included in the tenants’ HTTPS requests.

Initially, a zeroth instance of the application starts, warms up, creates and uploads the artifacts to the DCAS Service. Afterwards, the application is restaged, which triggers the compilation and droplet creation phase to be executed again. This is a crucial step; otherwise, the tenant’s droplet will not contain any artifacts by default and they would have to be downloaded again on every scale out from the artifact server. Instead, the first instance of the application downloads and packs the artifacts into the application’s droplet. Subsequent instances of the same application are started from this droplet, which now contains any dynamically compiled artifacts shared from the zeroth warm-up instance and packed into the droplet by the first instance. The extra restage required by DCAS should be considered an acceptable
tradeoff; applications are initially pushed during early production or late
development and at these times, SLO satisfaction is of a lesser concern.
However, spinning up scaled instances happens during load fluctuations that
threaten the violation of SLOs.
Furthermore, another issue to be taken into consideration is code updates
between application restages. In that case, DCAS identifies differences in the
code by using a checksum hash and discards previous artifacts if necessary.
A sequence chart describing the interactions of the different components of
the technique is displayed in Figure 5.1.

5.4.1 DCAS Service
The artifact server of the DCAS cloud Service provides the following func-
tionality: First, it accepts an application’s sharable artifacts and stores them
based on a provided checksum, overwriting any previous ones. Second, it
serves an application’s request for its sharable artifacts for a given checksum
and returns them, if available; returning appropriate error code otherwise.
Third, it stores the artifacts per the application’s unique cloud identifier
name as well as a value calculated by hashing the application’s source code.
Finally, it allows users to create accounts and manage their applications;
admin role users can manage all applications and other users.
Figure 5.1: A sequence chart outlining the various interactions in the proposed Dynamically Compiled Artifact Sharing (DCAS) technique.
5.4.2 DCAS Buildpack

Buildpacks are used in PaaS clouds to describe a set of required steps to create the image to be executed in a container. A DCAS buildpack needs to implement the following extra functionality. First, during the buildpack’s compile phase: The application’s checksum (to ensure the artifacts’ version matches with those stored remotely) is calculated by hashing the appropriate files. This can be done in various ways, depending on the frequency of change of the various components running in a container. Afterwards, a request is made to the DCAS Service to request this application’s dynamically compiled artifacts, using the calculated hash. If found, the artifacts are placed in a folder that will be packed into the droplet. Otherwise, the buildpack adds a special zeroth-startup script to the droplet folder.

Second, during the buildpack’s release phase: If the zeroth-startup script is found, it is set to start in the background alongside the main application. Also, the application is set up to produce dynamically compiled artifacts and store them in a specific location in its filesystem. Otherwise, the instance is set to start with the appropriate command line and environment settings required to point it to the existing dynamically compiled artifacts that are present in its droplet.

When an instance is eventually deployed on the cloud, if it runs in zeroth mode, the zeroth-startup script will wait for a certain amount of time and then submit any dynamically compiled artifacts to the DCAS Service using appropriate checksums and application-specific credentials. Otherwise, if
the application runs in normal mode, it will have its dynamically compiled artifacts ready to use and start faster.

DCAS was implemented on Cloud Foundry by modifying the IBM WebSphere Application Server Liberty Buildpack [18], which uses the IBM JVM [9] and the WebSphere Liberty application server [1], to abide by our DCAS buildpack design. curl\textsuperscript{3} was used for communicating with the DCAS Service and md5sum\textsuperscript{4} for hashing artifacts in order to produce versioning checksums.

5.5 Evaluation

Four Java EE tenant applications were used for experimentation, each including a varying number of JSP files (1, 2, 4, and 8) but overall performing the same amount of processing per request. Each request performed a number of CPU-intensive tight-loop calculations and object allocations using multiple Java classes and methods calling other methods; thus, forcing the JVM to create multiple classes for each application as well as to give room to the JIT to perform optimizations like method inlining and stack-instead of heap-allocation.

The nJSP applications were deployed on a local and isolated Cloud Foundry using the DCAS Liberty buildpack. The applications’ dynamically compiled artifacts from a first warm-up run were captured and uploaded on the DCAS service, which was also running outside Cloud Foundry, to minimize interfer-

\textsuperscript{3}https://curl.haxx.se/
\textsuperscript{4}https://linux.die.net/man/1/md5sum

85
ence. For the warm-up runs, multiple requests were fired to the applications while the zeroth-script was waiting to ensure that more experience is generated and exported before the generated artifacts were uploaded to the DCAS Service.

5.5.1 Scaling Performance

To experimentally measure any scaling improvements from DCAS, a testing script that performed the following tasks repeatedly for each testing application and buildpack (Default vs. DCAS) was developed: First, Cloud Foundry was cleared of any previous applications, if any. Second, the current application with the current buildpack was pushed with one instance using `cf push`. Third, using four JMeter threads, requests were fired for 200s. Fourth, a request to scale out the running application by one instance was made using `cf scale` and set to run in the background. Fifth and in parallel with the execution of the previous step, another 30s-long JMeter stress-test using two threads started. Sixth, the data collected from JMeter was post-processed and the startup time, warmup time, error rate and warmed-up response time of the scaled-out instance were extracted and appended to a results file. The startup time of the second instance was calculated via the instance’s first reply. The warmup time was defined as the first moment after startup that the scaled instance had a median response time 1% around that of the first instance for a sliding window of 32 responses.

The results, which are displayed in Figure 5.2 indicate that DCAS improved
Figure 5.2: Experimentally measured scale-out performance on a local and isolated Cloud Foundry installation comparing DCAS against the baseline.

the startup time by 3.1s (between 20% and 30% time reductions). However, there was no significant difference in warmup time in three out of four applications. Nevertheless, DCAS reduced the error rates by about half. Finally, in two out of four applications tested, the warmed-up response time was significantly lower in DCAS over Default but not in the other two. The results suggest that DCAS was able to significantly improve startup time and reduce response error rates. The reported JMeter errors occurred due to timeouts or bad responses, which were caused when an instance was not yet ready to serve a request or was overwhelmed by the incoming load.
5.5.2 Performance Interference

Recreating information that could be otherwise acquired through shared artifacts is resource intensive. As discussed earlier, this can cause resource consumption spikes, which in turn can interfere with the performance of co-located tenants in cloud deployments without hard resource limitations.

To test the effectiveness of DCAS in reducing performance interference on co-located tenants during scaling, a C++ web application was implemented. The C++ tenant runs a tight, CPU-intensive loop per request and was deployed on the same local and isolated installation of Cloud Foundry using a C++ community buildpack. C++ was used instead of Java or other high-level languages to minimize the measured performance variance; C++ is compiled directly to machine code instead of running on top of a runtime and does not implement automatic memory deallocation or just-in-time compiling, which can cause variant response times.

The C++ tenant was used as the foreground application and requests were fired at it using JMeter from a host outside the Cloud Foundry installation. Three types of tests were run multiple times to measure performance statistics of requests: First, Baseline, where the C++ tenant ran without interference; second, Default, where the C++ tenant served requests while a Java EE tenant using the default Liberty buildpack scaled out; and third, DCAS, where the Java EE scaling-out tenant used the DCAS buildpack instead of the default. In both Default and DCAS modes, the previously discussed nJSP Java EE applications were used for background tenants and
Figure 5.3: Experimentally measured throughput interference comparison between DCAS and the default mode that is imposed on another tenant running on the same local and isolated installation of Cloud Foundry.

were scaled out to add one and two extra instances at a time.

In Figure 5.3 and Figure 5.4 the experimentally acquired performance interference measurements are displayed, comparing the Default and DCAS modes over the no-contention Baseline. The average performance reductions over multiple runs are displayed; the error bars indicate the standard deviation (only throughput and the 90th and 99th percentile of response time for brevity; the other metrics were similar). In all cases, the performance interference of the DCAS mode was significantly lower than the Default. Additionally, the results indicate a significantly higher performance interference when two instances were scaled out at once versus only one. This is also expected because the CPU spikes of the two instances coincided and further reduced the available resources for the foreground, C++ tenant.
Figure 5.4: Experimentally measured response time interference—top for 90th and bottom for 99th percentile—comparison between DCAS and the default mode that is imposed on another tenant running on the same local and isolated installation of Cloud Foundry.

5.5.3 Hashing Time

As discussed earlier, the DCAS technique requires a version identification system to ensure that any changes in the application are detected and stale dynamically compiled artifacts are discarded. To this end, DCAS uses file hashing checksums and in this section we discuss the results of our measurements regarding the execution time to perform such operations. The same nJSP Java EE applications discussed earlier were used, which were deployed
Figure 5.5: Experimentally measured hashing time comparisons of various parts of the Liberty buildback container using the \texttt{md5sum} algorithm for four different Java EE applications.

on the same isolated Cloud Foundry installation as before using the Java EE Liberty buildback.

For each application, three types of hashing tests were conducted, depending on how much container information was included in the process. First, only the tenant application was hashed; second, the tenant application with the application server (Liberty Profile); and third, the tenant application, the application server and the runtime (IBM JVM). The rationale behind these three choices is that these three components are the main parts for this type of container; the tenant application is the most likely to be frequently updated, followed by the application server and finally the runtime. For the creation of the checksum, \texttt{md5sum} was used and for each condition, the hashing was repeated multiple times and its execution time was aggregated.

The average execution time of \texttt{md5sum} per different setup is displayed in Figure 5.5, using the standard deviation for error bars. The results, as expected, suggest that the more files are hashed, the longer the time to calculate the
checksum. Regardless, even if all of the major parts of the container were hashed for versioning, the whole process required less than 2.1s in all cases and as low as 0.22s when only the tenant application was hashed. Furthermore, this should be considered an acceptable tradeoff, since this task will happen only during the zeroth instance, which is not expected to handle a large number of requests.

5.6 Conclusions

Traditional ways of sharing language runtime experience among processes, such as shared memory, do not work in a straightforward way in the multitenant and distributed clouds, nor were they designed for such use. This inability to share dynamically compiled artifacts among cloud instances of the same scaled application results in two problems: First, the scaled instances themselves start up slower, which results in lower QoS and potential SLO violations as their responses might be too long or time out. Second, because of the clouds’ multitenant design, instances that start up without previously acquired experience require a resource-intensive phase in order to rebuild it. This results in a resource consumption spike, which in turn can affect and interfere with the performance of any co-located instances, if the cloud operates without fixed resource limitations.

The chapter outlined Dynamically Compiled Artifact Sharing (DCAS), a technique that enables the transparent and secure sharing of experience from
a first to subsequent scaled instances of applications running on buildpack-oriented PaaS clouds. DCAS uses an extra warm-up run from which dynamically compiled artifacts are collected and submitted to an external artifact service. Afterwards, the artifacts are downloaded during compiling the first instance of the application, and placed alongside its other artifacts in the application’s deployment package or droplet. Subsequently scaled instances are deployed from this droplet and thus have any dynamically compiled artifacts available. These artifacts were originally created for sharing among processes of the same OS only, but our technique overcomes this barrier.

The experimental evaluation, which was conducted on an isolated installation of the PaaS Cloud Foundry, measured significant improvements in startup time of scaled instances as well as error reduction of serving requests while scaling. Additionally, significant reductions in performance interference were also measured.

This work could be expanded on other runtimes and include further dynamically compiled artifacts. An interesting question is if long-term improvements can also be attained by DCAS—for example, by sharing profiling data.
Chapter 6

SLO Request Handling

Satisfying requests on time is a crucial aspect of cloud SLAs. However, application servers execute requests first-come-first-serve without considering SLO requirements. This chapter formally studies the on-time performance of application servers using an experimentally evaluated mathematical model. The proposed model is based on a certain number of connected clients instead of the traditional exponential arrival rate. Additionally, a request reordering technique is proposed and evaluated that uses previous execution-time data and the current load of the server to improve SLO satisfaction. Elements of this work have been published in the Proceedings of CASCON 2017 [93] and in a defensive publication by IBM for patenting purposes [94].
6.1 Motivation

Cloud applications commonly follow the client/server design pattern; the server-side runs on the cloud and remote clients make requests to it. SLAs require that a certain percentage of requests is satisfied within a certain amount of time, constituting the system’s SLO targets. Cloud applications are additionally designed following the microservices and stateless software patterns such that only the parts that receive the most stress are scaled. Thus, in-cloud requests can be produced by certain sets of connected services that are scaled out to a certain number of instances that communicate with each other. Additionally, external requests can cascade to a series of internal cloud requests across the various cloud services required for its completion. Consequently, on-time responses for all these components is crucial for maintaining acceptable response times to the users.

However, executing incoming requests First Come First Serve (FCFS) does not necessarily satisfy response-time SLOs. In addition, since cloud applications increase the number of server instances they use as a means to satisfy increased incoming load, a crucial problem of efficiency is being able to identify the ideal number of instances required such that the on-time SLO requirements of an application server are satisfied with the least amount of dedicated resources.
6.2 Related Work

Hu et al. proposed a mathematical model to describe service time and the lowest number of servers required to satisfy given SLOs [61]. However, that work operated under different assumptions regarding the characteristic of the incoming load, which it considered as arriving at a certain average rate over a given time-frame. Instead, this work focuses on modeling the load proportionally to the number of connected clients. Furthermore, the authors calculated an optimal number of servers to satisfy response-time SLOs using an iterative $O(n)$ approach for the calculation. In addition, they also studied the execution order of requests but did so with using two separate task-queues, each storing low- and high-priority requests respectively; instead, this chapter proposes reordering on the application-server level and based on projected SLO satisfaction regardless of priority.

A cost optimization method for IaaS clouds was proposed by Li et al. [77]. This method modeled the prices paid by both cloud provider and client on a per-resource manner. Additionally, it considered the QoS maintenance as a constraint while it aimed to reduce the reserved resources to the largest possible extent. The proposed method presupposed a performance model that predicted the effects on decision variables; elements of the present work could be integrated into such a model. Ghanbari et al. investigated optimal autoscaling for IaaS clouds [54]. The authors focused on satisfying the client’s SLOs while minimizing the number of leased VMs. Although related,
managing heavy-weight VMs for IaaS clouds is different to that of creating light-weight microservices instances or PaaS containers.

Work queues have been studied theoretically by Gross Harris [57]. In particular, specific nomenclature for queue theory was established and various types of queues were analyzed, such as simple and advance Markovian models, statistical interference, simulation and an example and theoretical case study. A theoretical study of an EDF queue was performed in [21]; however, an SLO-aware queue should be more suitable for cloud deployments.

Web systems have been proposed that, depending on execution times and deadlines provided by incoming requests, schedule them for execution, if there are available slots that do not violate their execution deadlines or denies their execution altogether [20, 19]. Also, similar work has been conducted on scheduling execution of longer tasks on grid computing systems [114]. However, this type of work requires very high-overhead micromanagement of the task queue and most importantly, assumes that execution times can be known a-priori or be part of the request. Even if an execution time can be associated with a specific type of request, performance interference depending on the current tenant’s load or the loads of co-located tenants can make these estimations invalid.

Various works proposed handling requests with known priorities using multiple task queues [134, 104] and including specific priority per file-type [86]. Nevertheless, they do not tackle the problem of all requests having the same priority but various execution times; additionally, they can suffer from star-
vation of low-priority requests and accurately sizing their thread-pools. Delaying the execution of requests was used to synchronize multimedia streaming that is spread out across multiple servers [49]. Furthermore, reordering tasks was investigated for network packets [23], hard disk read and write requests [76] and real-time operating systems [132].

Scaling in the cloud has been leveraged with triggers that activate the creation of new instances when particular resource consumptions reach certain thresholds. Services that automatically perform trigger-based scaling, either to the tenant application as a whole [116][51][50] or to its most stressed components [66], have been designed and included in cloud solutions. In this work, a mathematical equation that calculates the ideal SLO number of instances is provided, instead of the traditional trial-and-error and trigger-based techniques.

6.3 Modeling Satisfaction of SLOs

A cloud system serving remote clients can be modeled as a system of queues, each with multiple servers. A load balancer spreads requests on each server; the requests wait on the server’s task queue; and one of the server’s processing units eventually executes them and replies to the client. Assuming that the clients are performing a certain long-term stream of requests that require the previous one to complete, after a certain network or computational inter-request delay, the client fires back another request. Some clients eventually
leave and new clients come but for our model, we consider these events to be happening less frequently than the rate of requests; consequently, a system can be studied by the number of active clients it currently has.

Consider a cloud application server system that uses \( n \) servers, each with \( m \) processing units available utilizing a round-robin load-balancer and FIFO task-queues. Also, assuming that the requests’ execution time is normally distributed with average \( \mu \) and standard deviation \( \sigma \); \( k \) clients are cycling through the system; and the inter-request delay is considered negligible. Therefore, the average response time \( R \) of a request will be equal to the sum of the average waiting time \( W \) and the average processing time \( \mu \):

\[
R = W + \mu
\]  

(6.1)

A new request will need to wait on the queue for \( r \) waiting processing rounds until all previous requests have been executed:

\[
r = \left\lceil \frac{k - m \cdot n}{m \cdot n} \right\rceil
\]  

(6.2)

The waiting time will either be 0, if the number of clients is less than or equal to the total number of processing elements, or proportional to the expected number of processing rounds required to execute enough tasks ahead in the queue. Thus, the waiting time is normally distributed and calculated as the sum of multiple rounds of the normally distributed processing times of the
$$W = \sum_{i=1}^{r} \mu = r \cdot \mu = \max \left( \frac{k - m \cdot n}{m \cdot n} \cdot \mu, 0 \right) \quad (6.3)$$

Therefore, the average response time is also normally distributed with average response time $R$:

$$R = W + \mu = \mu + r \cdot \mu = \mu \cdot (r + 1) = \max \left( \frac{k}{m \cdot n} \cdot \mu, \mu \right) \quad (6.4)$$

Furthermore, statistical error propagation for sums of independent variables is calculated as follows:

$$\sigma_{A+B} = \sqrt{\sigma_A^2 + \sigma_B^2} \quad (6.5)$$

Consequently, because the response time is essentially a sum of multiple processing rounds, each with standard deviation $\sigma$, the overall standard deviation $R_\sigma$ of the response time is:

$$R_\sigma = \sqrt{\sum_{i=1}^{r+1} \sigma^2} = \sigma \cdot (r + 1) = \max \left( \sqrt{\frac{k}{m \cdot n} \cdot \sigma}, \sigma \right) \quad (6.6)$$

The throughput $T$ of the system is directly connected to the maximum number of requests that can be handled, which is either the number of clients or the total number of processing units, divided by the average response time:

$$T = \frac{\min(k, m \cdot n)}{\mu} \quad (6.7)$$
An SLI metric for evaluating the response time SLO satisfaction for a certain SLO time limit is the percentage of requests that are completed on time. This is equivalent to the probability a request has a response time less than or equal to the SLO time:

\[ \text{OnTime}\% = P(R \leq \text{SLOTime}) \quad (6.8) \]

However, because the response times are normally distributed, this probability is calculated by the cumulative normal distribution function \( D_{R,R,\sigma} \) (Figure 6.1):

\[
P(X \leq x) = D_{\mu,\sigma}(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\sigma^2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx \quad (6.9)
\]

Thus:

\[ \text{OnTime}\% = D_{R,R,\sigma}(\text{SLOTime}) \quad (6.10) \]

Also another SLI metric is the on-time throughput of the system. It is defined as the rate at which requests are completed on time; therefore it is equal to the product of the on-time percentage with the throughput of the system:

\[ \text{OnTimeThru} = \text{OnTime}\% \cdot T \quad (6.11) \]

Finally, another useful SLI is that of the minimum time required for a given proportion of requests to respond. This is the nth percentile of the response
time and for example, the 95th percentile would be the minimum time required for 95% of the requests to respond. This can be derived from the quantile function $Q_{R,R_o}$ describing the normal distribution of the responses (Figure 6.1), which is the inverse of the cumulative function $D_{R,R_o}$:

$$Q_{R,R_o}(x) = D_{R,R_o}^{-1}(x)$$

Consequently:

$$\text{Resp}_n = Q_{R,R_o}\left(\frac{n}{100}\right)$$

### 6.3.1 Theoretical Model Validation

To evaluate the proposed model, a Java EE application was implemented using a Servlet that blocks on a semaphore for a randomly generated and normally distributed amount of time, controlled by input parameters.
Figure 6.2: Comparison of the theoretical model predictions on the left versus the experimentally acquired measurements run on a local and isolated Cloud Foundry installation on the right. (Part 1/2).

Experiments were conducted with three combinations of average processing time and standard deviation, with multiple repetitions and number of clients. A local and isolated installation of Cloud Foundry using the Java Liberty buildpack was utilized. The number of semaphore permits was set to 8, which represents a system with that many processing elements. The experimental values of the SLIs were captured and aggregated; the theoretical SLI values were also calculated using the proposed model. Comparing the experimental and theoretical results suggests that the proposed model closely matched the real values. Small deviances were also registered, particularly in predicting the average response time; but overall, the
Figure 6.3: Comparison of the theoretical model predictions on the left versus the experimentally acquired measurements run on a local and isolated Cloud Foundry installation on the right. (Part 2/2).

model should be considered a good approximation of what happens in reality. Graphs comparing the results are displayed in Figure 6.2 and Figure 6.3.
6.4 SLO Number of Instances

A straightforward application of the theoretical model is to decide the minimum number \((n)\) of application servers per number of parallel clients such that the on-time satisfaction rate is maintained above a certain SLO threshold SLOThres, which is set by the SLO requirements of the application. Therefore, using the model, solving the following equation for the smallest integer value of \(n\) should provide the answer:

\[
\text{OnTime}\% = \text{SLOThres} \quad (6.14)
\]

Using Equation 6.10, the following is deduced:

\[
D_{R,R_\sigma}(\text{SLOTime}) = \text{SLOThres} \quad (6.15)
\]

The cumulative probability function \(D\) of the normal distribution can be calculated using the Gauss error function \(\text{erf} [22]\):

\[
D_{R,R_\sigma}(\text{SLOTime}) = \frac{1}{2} \left( 1 + \text{erf} \left( \frac{\text{SLOTime} - R}{R_\sigma \sqrt{2}} \right) \right) \quad (6.16)
\]

Therefore, Equation 6.14 is equivalent to:

\[
\frac{1}{2} \left( 1 + \text{erf}(Y) \right) = \text{SLOThres} \quad (6.17)
\]
With
\[ Y = \frac{\text{SLOTime} - R}{R\sigma\sqrt{2}} \]  
\hspace{1cm} (6.18)

A commonly used approximation [22] for the erf function is:
\[
erf(x) \approx \text{sgn}(x) \sqrt{1 - \exp\left(-x^2 \frac{\frac{4}{\pi} + ax^2}{1 + ax^2}\right)}, \quad a = 0.147 \]  
\hspace{1cm} (6.19)

Now, in order to solve this equation for \( n \), two cases are considered: First, \( \text{SLOThres} \geq 0.5 \) and second, \( \text{SLOThres} < 0.5 \):

**Case 1:** \( \text{SLOThres} \geq 0.5 \Rightarrow Y \geq 0 \Rightarrow 
\[
\frac{1}{2} \left( 1 + \sqrt{1 - \exp\left(-Y^2 \frac{\frac{4}{\pi} + aY^2}{1 + aY^2}\right)} \right) = \text{SLOThres} \]  
\hspace{1cm} (6.20)

Which is equivalent to:
\[
aY^4 + (aL + \frac{4}{\pi})Y^2 + L = 0 \]  
\hspace{1cm} (6.21)

With
\[
L = \ln(4\text{SLOThres} - 4\text{SLOThres}^2) \]  
\hspace{1cm} (6.22)

Equation 6.21 is a square quadratic; therefore, it can be solved as a simple quadratic but by keeping only the positive solution:
\[
\Delta Y^2 = (aL + \frac{4}{\pi})^2 - 4aL \]  
\hspace{1cm} (6.23)
And
\[ Y^2 = \frac{-(aL + \frac{4}{\pi}) + \sqrt{\Delta Y^2}}{2a} \] (6.24)

The next step in this calculation is to go back to Equation 6.18 and apply the results of Equation 6.24:
\[ Y^2 = \left( \frac{\text{SLOTime} - R}{R \sigma \sqrt{2}} \right)^2 \] (6.25)

Also, the values of \( R \) and \( R \sigma \) from Equations 6.4 and 6.6 can be replaced. For brevity, the \( k > m \cdot n \) case is used, i.e. there are more clients on the system than the total number of processing units:
\[ Y^2 = \left( \frac{\text{SLOTime} - \frac{\mu k}{m \cdot n}}{\sigma \sqrt{\frac{k}{m \cdot n} \sqrt{2}}} \right)^2 \] (6.26)

Resulting in:
\[ \text{SLOTime}^2 n^2 - \left( 2\text{SLOTime} \frac{\mu k}{m} + 2Y^2 \frac{\sigma^2 k}{m} \right) n + \frac{\mu^2 k^2}{m^2} = 0 \] (6.27)

This is also a quadratic equation; thus:
\[ \Delta_n = \left( 2\text{SLOTime} \frac{\mu k}{m} + 2Y^2 \frac{\sigma^2 k}{m} \right)^2 - 4\text{SLOTime}^2 \frac{\mu^2 k^2}{m^2} \] (6.28)
And finally, because $n$ also needs to be rounded up to the closest integer:

$$
n_1 = \left\lceil \frac{2\text{SLOT} \mu_k m + 2Y^2 \sigma_k^2 m + \sqrt{\Delta_n}}{2\text{SLOT} \mu m^2} \right\rceil \tag{6.29}
$$

Only the positive root solution was kept from this quadratic equation; the negative one violates the SLOThres $\geq 0.5$ condition and is discarded. 

**Case 2:** SLOThres $< 0.5 \Rightarrow Y < 0$ Similarly to Case 1:

$$
n_2 = \left\lceil \frac{2\text{SLOT} \mu_k m + 2Y^2 \sigma_k^2 m - \sqrt{\Delta_n}}{2\text{SLOT} \mu m^2} \right\rceil \tag{6.30}
$$

Consequently,

$$
n = \begin{cases} 
n_1, & \text{if } \text{SLOThres} \geq 0.5 \\
n_2, & \text{if } \text{SLOThres} < 0.5
\end{cases} \tag{6.31}
$$

Equation 6.31 can be used by a cloud system to determine the lowest number of scaled-out instances required to be deployed on the provider’s clusters for its client/server applications to maintain their on-time response percentage below the SLO stated threshold based on their current load. This $O(1)$ decision can be triggered at the router or load balancer level, which handles the incoming throughput and can infer an application’s load.

---

1This is actually the negative root solution of Case 1 because of the odd symmetry of the erf function: erf($x$) = -erf(-$x$)
6.4.1 Experimental Validation

In order to validate the theoretical predictions, the experimental setup described in Section 6.3.1 was appropriately expanded. In particular, the calculation of Equation 6.31 was added in the testing script and the Cloud Foundry command `cf scale -i`, which scales the application out to the requested number of instances, was invoked accordingly. Multiple experiments were conducted, for 1 to 100 parallel clients, using three combinations of $\mu, \sigma$ processing times and three SLO thresholds. The theoretically predicted and experimentally acquired on-time percentages are compared in the charts of Figures 6.4, 6.5 and 6.6.

The experimentally acquired results generally match the predicted ones; however, sometimes the experimental results fared worse than the prediction. This discrepancy can be attributed to the occasional failure of spinning up new instances, which was discovered by studying the Cloud Foundry logs. This explanation further matches the experimental results due to the increased likelihood of an SLO violation just before a scale-out was required when the system was under increased stress. Using smaller processing times leads to more frequent requests; lowering SLO thresholds causes instances to handle more clients concurrently; and both of these factors contributed to further stressing the system. Finally, the effects of the GC and other runtime components as well as network delays, which were excluded from the model, might have contributed as these effects become more significant during times of higher stress.
Figure 6.4: Comparison of theoretical and experimentally acquired results (run on a local and isolated installation of Cloud Foundry) of maintaining the OnTime% above a certain SLO threshold for a varying number of parallel clients and $\mu = 0.5, \sigma = 0.25$. 
Figure 6.5: Comparison of theoretical and experimentally acquired results (run on a local and isolated installation of Cloud Foundry) of maintaining the OnTime% above a certain SLO threshold for a varying number of parallel clients and $\mu = 0.25, \sigma = 0.25$.
Figure 6.6: Comparison of theoretical and experimentally acquired results (run on a local and isolated installation of Cloud Foundry) of maintaining the OnTime% above a certain SLO threshold for a varying number of parallel clients and $\mu = 0.5, \sigma = 0.1$. 

112
Table 6.1: Root mean square error calculations between theoretically predicted and experimentally acquired results that illustrate the efficacy of the proposed model.

<table>
<thead>
<tr>
<th>SLO Threshold / Processing Time</th>
<th>$\mu = 0.5 , \sigma = 0.25$</th>
<th>$\mu = 0.25 , \sigma = 0.25$</th>
<th>$\mu = 0.5 , \sigma = 0.1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>90%</td>
<td>2.03%</td>
<td>1.93%</td>
<td>3.05%</td>
</tr>
<tr>
<td>60%</td>
<td>3.04%</td>
<td>5.75%</td>
<td>10.76%</td>
</tr>
<tr>
<td>30%</td>
<td>6.64%</td>
<td>7.36%</td>
<td>5.44%</td>
</tr>
<tr>
<td>GeoMean</td>
<td>3.45%</td>
<td>4.34%</td>
<td>5.63%</td>
</tr>
</tbody>
</table>

The root mean square differences between experimental and theoretical values were calculated and are displayed in Table 6.1. These errors quantify the accuracy of the model in predicting realistic SLO satisfaction on scaling.

### 6.5 SLO Request Execution

Improving the SLO satisfaction can happen through reordering the execution of requests assuming the server can predict the expected execution time of each request. On one hand, during low loads, requests that will take a long time to execute need to be given priority within SLO response time limits, which will ensure that they respond on-time. On the other hand, once the load is too high, the server is doomed to not respond on time; and in that case, it is preferable for short-running requests to be executed with priority such that at least some of them finish on time. Such a load-based, execution time prediction-based technique for application servers is now proposed and evaluated.

The proposed technique works as part of a client/server architecture and in
A new request arrives and always handled by the main thread

The incoming connection is accepted

A new SLAHandlingRequest task is pushed to the SLA Pool to handle this request

The SLA Pool looks up the previous execution time of an SLAHandlingRequest task in the hash table

The new task’s execution startup deadline is calculated

The SLA Pool inserts the new task to its priority queue, which reorders it based on the calculated deadline

Repeat

The threads of the SLA Pool are initialized and block on the priority queue waiting for a task to execute

A new task is retrieved from the queue

Measure current time

Based on the type of the task

SLAHandlingRequest

Servlet

The path field is read from the HTTP request

Execute Servlet’s run function

The Servlet that needs to be executed is found by looking up the extracted path

A new task is created of type that of the requested Servlet and it is pushed in the SLA queue

Measure current time again and update the execution time of this type of task in the hash table

Repeat

Figure 6.7: Two diagrams illustrating the proposed algorithm of SLO request reordering. The diagram on the left describes the actions to be performed when a new request arrives on the applications server; whereas, the diagram on the right, outlines the algorithm of the threads in the SLA pool.
particular, it handles incoming requests in an SLO-aware way, by reordering their execution based on their previous execution times. There are three main parts: first, a load monitoring daemon; second, the SLO Task Queue; and third, the SLO Request Reordering. The overall architecture is displayed in Figure 6.7.

6.5.1 Load Monitoring Daemon

When the application server begins execution, a background daemon thread is started that periodically reads the network I/O traffic of the current container. The daemon thread updates an internal variable with the most up-to-date value of the input network throughput, which is subsequently accessed in a thread-safe and fast way by the SLO Task Queue.

6.5.2 SLO Task Queue

The SLO Task Queue is a data structure that manages the execution of incoming tasks, potentially in parallel. It is initialized with the number of threads it can use and the SLO response time its tasks need to abide by. Furthermore, the SLO Task Queue maintains a Concurrent Hash Table with the most recent execution time of a task per task-type. The type of the task is determined by the class of its executable; although, this could be easily changed to be part of the path of an HTTP request or part of the incoming WebSocket message.
When a new task is added, its expected (Expected) execution time is looked up on the most recent execution time hash table. If no entry is found, the expected execution time is conservatively set to the SLO response time. Two cases are considered:

First, during low load, the execution deadline of the incoming task, which determines its order in the queue, is calculated as the latest possible time that the task can start its execution and still abide by its SLO requirement:

\[
\text{Deadline}_{\text{Low}} = \text{now} + \max(SLO\text{Time} - \text{Expected}, 0) \quad (6.32)
\]

Second, during high load, the execution deadline is calculated at a moment into the future proportional to the task’s expected execution time but bounded by the SLO response time.

\[
\text{Deadline}_{\text{High}} = \text{now} + \min(\text{Expected}, SLO\text{Time}) \quad (6.33)
\]

Depending on a threshold value, the SLO Task Queue decides whether or not the server is currently experiencing low or high load and appropriately assigns an execution deadline for the incoming task. The load threshold can be either user-defined or automatically approximated through a machine-
learning algorithm that aims to optimize SLO satisfaction.

\[
\text{Deadline} = \begin{cases} 
\text{Deadline}_{\text{Low}}, & \text{if thru} \leq \text{Threshold} \\
\text{Deadline}_{\text{High}}, & \text{Otherwise}
\end{cases} 
\]  

(6.34)

Afterwards, the task and its latest execution time are wrapped around a special Task-Wrapper pushed into a priority queue; the tasks are ordered by ascending latest execution time.

**Algorithm 1** SLO Task Queue Execute Task

```
procedure execute(task)
    expected ← previousResp.get(task.getClass)
    if expected = null then
        expected ← SLOResp
    end if
    if thru ≤ Threshold then
        deadline ← now() + max(SLOTime − Expected, 0)
    else
        deadline ← now() + min(Expected, SLOTime)
    end if
    taskQueue.add(new WrappedTask(task, deadline))
end procedure
```

Additionally, the Task-Wrapper objects measure the execution time of the task and update an execution-time hash map after a task completes. The key can be either the class type of the task, the full (or part) of the request HTTP path or part of the WebSocket message that initiated the request. The hash map is updated with the weighted average of its current value and the execution time of this run, although other machine learning techniques
could be used instead:

\[
\text{Expected}_n = a \cdot \text{Expected}_{n-1} + (1 - a) \cdot \text{ExecutionTime}_n
\]  \quad (6.35)

Finally, the thread(s) of the SLO Task Queue are repeatedly assigned wrapped tasks from the queue and execute them. The queue is a blocking one; therefore, when a thread does not find any task to execute, it does not spin, instead it is just blocked until some task becomes available. The details of the task insertion method of the SLO Task Queue in Algorithm 1 and of the Task-Wrapper run method are displayed in Algorithm 2.

**Algorithm 2 SLO Task Queue Wrapped Task**

```
procedure run
    t0 ← now(); task.run(); dt ← now() − t0
    previousResp.put(task.getClass, dt)
end procedure
```

### 6.5.3 Reordering

Each incoming request is handled in three stages: First, the main thread of the application server accepts a new client, creates a new Request-Handler task with the request’s connection data and pushes it into the SLO Task Queue. Second, when the Request-Handler task is scheduled for execution, it reads the first line of the HTTP message in the request and depending on the requested path, it looks up which task-type is required for this request, creates this task with the request’s connection data and pushes it into the
SLO Task Queue. For WebSocket implementations, after the connection is upgraded, new incoming messages are the ones initiating new tasks. Finally, the task is popped from the queue and executed.

The algorithm of the first two stages is displayed in Algorithms 3 and 4 respectively. The third stage is omitted because it depends on the tenant running on the application server.

Algorithm 3 SLO Request Handling Stage 1

```
procedure SERVER_THREAD
  loop
    reqHandler ← new ReqHandler(server.accept())
    threadPool.execute(reqHandler)
  end loop
end procedure
```

Algorithm 4 SLO Request Handling Stage 2

```
procedure RUN
  path ← extractPath()
  servletClass ← routesMapping(path)
  servletObject ← new servletClass()    ▶ Reflection
  threadPool.execute(servletObject)
end procedure
```

6.5.4 Experimental Evaluation

In order to evaluate the technique, a Java application server that supports a subset of the Java EE specifications was implemented. Five different modes for handling the incoming requests were compared: Baseline, where new
requests were pushed into a JUC Executor Pool and handled in a single-stage in FCFS order; Split-Only, which also uses a FIFO Executor Pool but handles the requests in three-stages; SLO Aware, which implements the proposed technique; Long-First, which prioritizes longest tasks first; and Short-First, which always prioritizes shortest tasks.

A test application with four servlets was created, each with exponentially different levels of computations. Experiments were conducted multiple times with various loads using all modes and an SLO response time of 1s. Randomly-selected requests were repeatedly fired, using the same random seed.

In Figure 6.8, the SLO aware technique is compared against the Baseline. As theoretically expected, there was no significant difference in average response time and throughput. However, there were measurable improvements in the on-time satisfaction and on-time throughput for medium and high loads. Additionally, the 95th percentile of the response time improved for low loads but deteriorated for high loads. Nevertheless, the goal is to respond on-time; assuming that requests are satisfied independently, adding a small delay to requests that are anyway failing in order to allow others to respond on time is a viable tradeoff.

We also compared the baseline with the Split-Only method but just splitting the execution of the tasks provided only small throughput improvements. Additionally, partially applying the SLO reordering technique does not provide the full benefits as the measurements (Figure 6.9) reveal: the Short-First mode fails to timely satisfy low loads; whereas, the Long-First mode fails to
Figure 6.8: Experimentally acquired results on a local and isolated installation of Cloud Foundry comparing the performance of the proposed SLO mode against the baseline.

The per number of clients results of all modes were further aggregated and are displayed in Table 6.2, color-coded green for better and red for worse. More specifically, the SLO mode improved the on-time satisfaction rate 1.81 times over the baseline; the Split-Only was marginally better than the Baseline by 1.01 times. Additionally, similar results were acquired for the on-time
Figure 6.9: Experimentally acquired results on a local and isolated installation of Cloud Foundry comparing the performance of the proposed SLO mode against the discussed Long-First and Short-First modes.

throughput. On average there was a 7.33% higher percentage of on time satisfaction between SLO Reordering and Baseline. Furthermore, the SLO threshold of 95% was satisfied by 10 different numbers of clients for the SLO mode versus 8 for Baseline. The performance decreased by 1% between SLO Reordering and Split-Only; this should be attributed to the overhead of using a priority queue. Nevertheless, because the aim is increasing the on-time response satisfaction, this should be considered an acceptable tradeoff.
Table 6.2: Aggregate experimental results grouped by various numbers of parallel clients. Green shading means improvement; whereas, red means worsening in comparison to the baseline.

<table>
<thead>
<tr>
<th>Ratio Over Baseline</th>
<th>Baseline</th>
<th>Split Only</th>
<th>SLO</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnTime%</td>
<td>1.00</td>
<td>1.01</td>
<td>1.81</td>
</tr>
<tr>
<td>OnTimeThru</td>
<td>1.00</td>
<td>1.03</td>
<td>1.81</td>
</tr>
<tr>
<td>Thru</td>
<td>1.00</td>
<td>1.02</td>
<td>1.81</td>
</tr>
<tr>
<td>Resp95</td>
<td>1.00</td>
<td>0.98</td>
<td>0.97</td>
</tr>
<tr>
<td>AvgResp</td>
<td>1.00</td>
<td>0.98</td>
<td>0.99</td>
</tr>
</tbody>
</table>

OnTime% Difference: 0.00% 0.12% 7.33%

#Loads OnTime% >95%: 8 9 10

<table>
<thead>
<tr>
<th>Ratio Over Baseline</th>
<th>Long First</th>
<th>Short First</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnTime%</td>
<td>0.99</td>
<td>1.57</td>
</tr>
<tr>
<td>OnTimeThru</td>
<td>1.01</td>
<td>1.60</td>
</tr>
<tr>
<td>Thru</td>
<td>1.02</td>
<td>1.02</td>
</tr>
<tr>
<td>Resp95</td>
<td>0.95</td>
<td>1.21</td>
</tr>
<tr>
<td>AvgResp</td>
<td>0.99</td>
<td>0.99</td>
</tr>
</tbody>
</table>

OnTime% Difference: -0.08% 5.14%

#Loads OnTime% >95%: 10 4

6.5.5 Discussion

The proposed technique targets applications that must respond within a given time limit and otherwise, their responses are no longer valid or timeout. Additionally, this technique is suitable for systems that allow performance interference; thus, applications whose response times can vary depending on the current load. The default FCFS approach falls short regarding on-time request satisfaction as the server becomes saturated. This is a crucial point in the management of a scaled out cloud application as the provider will be rushing to administer new instances to accommodate the increased load. Providing small on-time improvements can reduce cascading SLO failures. Additionally, due to performance interference present in these systems, it is preferable to measure request execution times per instance in order to more...
effectively reorder them. The alternative is to conduct this at the load bal-
ancer level. However, this is not optimal as the various instances of the same 
application might be co-located with other tenants that produce significantly 
different levels of noise. Consequently, the proposed SLO reordering idea is 
more applicable at the container middleware level.

6.6 Conclusions

Client/server architectures are heavily used in cloud computing, for both 
serving the users directly and indirectly through secondary services the pri-
mary server requires to communicate with. This enables application servers 
to be stateless, which in turn allows them to easily match the incoming load 
of requests by scaling out, i.e. adding more server instances.

The various applications that run on a cloud system have to abide by certain 
SLOs that include performance and on-time response limitations. Abiding by 
these performance SLOs is crucial for ensuring acceptable waiting times for 
the end user, especially since a request might actually require to be broken 
down and served by multiple cloud services. Also, SLOs carry with them 
financial benefits and penalties; therefore, they cannot be easily disregarded. 
Therefore, the design of application servers and load balancers must take into 
account SLOs and not just focus in maximizing their throughput or mini-
mizing their response time. To this end a theoretical model of SLO request 
satisfaction was presented that mathematically describes this behavior. The
model targets loads that are produced by agents, located internally that have the capacity to respond immediately after their request is satisfied. Thus, it is based on the number of connected clients a cloud service is handling at a given moment instead of using average arrival rates. The model is also used to calculate the lowest number of instances required for a cloud server to abide by its on-time response threshold per different numbers of connected clients. The theoretically derived equation was also experimentally validated. In addition, an SLO reordering technique for application servers was designed, implemented and evaluated on a minimal Java EE application server with measurable improvements on the on-time metrics when compared to the baseline FIFO. More specifically, in low loads the execution of longer requests was prioritized; whereas for high loads, the priority was given to shorter requests.

Future work includes expanding the model for a collection of interconnected services with individualized SLOs; experimenting with different machine learning algorithms regarding the request reordering technique; and incorporation of further SLO requirements. Finally, it should also be explored and incorporated on the application server, whether failing SLOs instead of overprovisioning resources is preferable according to cost models [48].
Chapter 7

Garbage Collection SLOs

Garbage collection can have a potentially deleterious effect in the on-time satisfaction of cloud requests, which could result in SLO violations. In this chapter, a testing methodology and a versatile and highly configurable benchmark that explicitly stresses the GC component of the runtime are presented. Using these tools, the SLO performance of the four GC policies of the IBM JVM is tested. The results suggest that the default GenCon policy is the most performant for the vast majority of the testing cases. Elements of this work have been previously published in ICACON 2016 [91] and IEEE CLUSTER 2017 [92]; an extended version is scheduled to appear in the International Journal of Cloud Computing [95].
7.1 Motivation

The convenience offered by high-level language runtimes with regard to automatic memory deallocation comes at a price: the GC component can be a source of bottlenecks, contention and performance degradation. Furthermore, a significant impact on SLO satisfaction can manifest on cloud applications, if the GC affects the performance of the tenant in unpredictable ways. This is the case because different schemes of managing the heap can result in various ways that the GC threads interfere and compete with the tenant’s code.

In addition, because of synchronization effects and in particular, to avoid race conditions that can invalidate the application’s data and execution patterns, GC algorithms frequently require the tenant application to completely halt its execution while certain types of memory management operations are conducted. This phase is referred to as Stop-the-world and it can be a crucial factor for causing timeouts in responses [65].

However, benchmarking of cloud applications has focused on the transactions of the application with cloud services, like databases, and does not emphasize stressing the GC. Consequently, refocusing the attention of the cloud community on the usually transparent memory management operations but potentially threatening SLA satisfaction is essential.

Interesting research questions that could be tackled with a GC-oriented cloud testing framework include the breadth of its parameters in terms of SLO
satisfaction and runtime stress; finding the ideal GC policy in terms of SLO satisfactions when running on a cloud slice with and without performance interference; and which GC policy affects the least the SLOs of any co-located tenants.

7.2 GC Algorithms

One approach to conducting GC is Mark&Sweep. A GC is triggered when an allocation of an object cannot be completed because there is not enough memory on the heap or a certain heap utilization threshold has been crossed. The GC threads then start by marking all objects reachable from the root sets recursively. Afterwards, all the heap is swept, any unmarked objects are considered dead and the memory they occupied is freed—usually, this does not mean that the memory is returned to the OS; instead, the runtime still holds it to be used on a future allocation [65] [27].

Mark&Sweep called repeatedly can eventually cause the heap to become fragmented because sweeping never moves live objects and the memory space between them can become perforated with holes of various sizes. Mark&Compact GC techniques have been proposed to solve this problem and they generally move live objects closer together after the marking phase [65] [27].

A second solution to the fragmentation problem is Copying GC, which splits the available heap into two parts: a from-space and a to-space. Objects are always allocated on the from-space and when it becomes full or its utilization

128
crosses a threshold, the GC threads instead of marking live objects, they directly evacuate them to the to-space. After the GC completes, any objects left on the from-space are automatically garbage and the whole chunk is reclaimed at once by swapping roles between the from- and the to-space. However, only half of the heap is available for allocations at any given point and GCs will happen twice as often [65] [27].

GC threads can also work alongside mutators: for instance, Concurrent Marking phases have been proposed for Mark&Sweep, which enable a big chunk of the GC work to be amortized between the Stop-the-World phases. However, sweeping is still conducted all at once and big delays can still occur; especially, if the heap size is large enough, such as in the case of in-memory databases [65] [27].

Research in the field suggests that objects tend to die young, an idea that matured into Generational Collectors that split the heap into multiple generations based on how many GCs they have survived. Thus, more frequent GCs can happen on the younger generations, while collecting older ones is deferred as much as possible [65] [27].

Finally, another approach of amortizing GC costs across time (and for our case, across requests) is Region-Based Collection that split the heap into numerous regions (currently in the order of a few thousands) and then manage them separately. Region-Based Collectors can manage each region using a combination of the aforementioned techniques: for example, the regions might be split into ages as in the Generational Collectors; young regions
Table 7.1: GC Policies of the IBM J9 JVM, which runs on OMR, and their corresponding features.

<table>
<thead>
<tr>
<th>Features</th>
<th>Policies</th>
<th>GenCon</th>
<th>Balanced</th>
<th>OptAvgPause</th>
<th>OptThruPut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compaction</td>
<td>Tenure</td>
<td>MC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Stop-the-World</td>
<td>✓</td>
<td>Global</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Concurrent Mark</td>
<td>✓</td>
<td>GMP</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concurrent Sweep</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generations</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Regions</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partial GCs</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

could be collected with *Copying GC*; older regions with *Mark&Compact*; and when necessary, an overall *Mark&Sweep* can clean up any remaining dead objects [65] [115].

### 7.2.1 GC Policies

Java runtimes have gained significant popularity not only because they can run Java but also, a number of other JVM languages, such Ruby, Python and Node.js. IBM’s runtime framework OMR\(^1\), which was recently distilled from IBM’s production-level J9 JVM [9], offers four GC policies: *GenCon*, *Balanced*, *OptAvgPause* and *OptThruput*. The features of these four policies are summarized in Table 7.1 [27] [115] [100].

In particular, *GenCon*, the default policy, is a *Generational* and *Concurrent* collector. It uses copying for its nursery, *Mark&Compact* for its tenured space and whenever possible, performs marking concurrently to the mutator threads’ execution. It aims to use the generational hypotheses and thus, minimize the proportion of objects that ever become tenured. Its frequent

\(^1\)https://projects.eclipse.org/proposals/omr
and small GCs are expected to amortize GC delays across time.

*Balanced* is a *Generational, Region-Based Collector*. It performs frequent *Partial GCs* using a subset of the regions whose alive objects are evacuated with *Copying*; and if necessary, full-heap GC, including the use of a *Concurrent Global Marking* and *Mark&Compact*. Its main design goal is to minimize pause times, which is achieved by not requiring large stop-the-world phases to scan a very large heap; instead, only a small subset of regions are scanned. *OptAvgPause* is a concurrent *Mark&Compact*; whereas *OptThruput*, is a classic collector. These two GC policies simplify and minimize the work that the GC performs such that the overall throughput of the application is maximized. As a result, they are more suited for solving long tasks whose execution normally includes multiple GC cycles over time.

### 7.3 Related Work

Cloud runtimes support high-level programming languages, like Java, Node.js, Python, PHP and Ruby. High-level languages provide automatic memory deallocation; runtimes implement this functionality through GC. However, GC is an expensive task and can lead to slowdowns, bottlenecks and performance degradation [65].

Two main types of Java benchmarks focus on either computational and memory-intensive tasks or client/server design with database transactions. The former target Java Standard Edition (SE); whereas the latter and most
relevant to cloud computing, target Java Enterprise Edition (EE). Java SE benchmarks include the DaCapo Benchmark Suite [32] as well as SPEC CPU and SPECjbb. Java EE benchmarks include CloudTrader, TPC-W [82] and a multitenant version of TPC-W [75]. However these Java EE benchmarks focus on transactions and database communication, neither targeting nor stressing the GC component of the runtime.

Related Java cloud work also includes: identifying potential memory leaks in cloud applications using data acquired by bytecode instrumentation and JVMTI [122]; collecting and aggregating GC log information from multiple cloud nodes [38]; an automated GC tuning tool for Java MapReduce on clouds [119]; and a tool for analyzing GC performance on the cloud by examining GC logs of Java applications distributed across nodes [70].

Finally, work has also been done on the C/C++ underlying level: A tool that simulates dynamic memory management has been implemented [84] and later expanded with an automated trace-file generator [105]. The proposed CloudGC was inspired by the graph model and operations presented in these papers; however, CloudGC is not a C/C++ memory simulator, but an actual Java EE application that allocates Java objects, stressing a real GC.

7.4 CloudGC

CloudGC is a GC-intensive and request-based application suitable for deployment as a cloud service. It is highly parameterized to enable the creation
of various memory allocation and access patterns, essentially causing broad ranges of pressure the mutator cloud-service instance can exert on the runtime and its GC. The design goals of CloudGC is to be used by research and industry to stress-test runtimes, deployment architectures, GC policies and GC algorithms.

7.4.1 In-Memory Object-Graph

CloudGC stresses GC by maintaining large numbers of interconnected objects on the runtime’s heap. A set of objects is allocated and made to point to each other during an initialization phase. This set is stateful and is accessible to the Servlet code satisfying the requests. Afterwards, each request performs a number of graph operations using as a starting point the initial set, but also working on its own set of objects that potentially do not escape to the stateful layer and survive only during the execution of this request. Each request performs a number of graph actions on the object graph such as allocation and reference change in order to stress the GC.

7.4.1.1 Stack Frames

A set of objects are directly accessible from the thread stacks and are referred to collectively as the rootset. Each thread stack is organized into stack frames; each stack frame represents an instance of a method invocation. Stack frames hold a number of object pointers to root objects and when the function that implements them returns, all references in this stack frame are
Figure 7.1: CloudGC organizes its memory in a stack frame architecture to replicate the way real-life applications work. Essentially, this design and based on the configuration parameters enables the existence of cold long-living objects, hot short-living objects, objects dying *en-masse*, objects escaping their allocation thread, etc.

released *en masse*. When a method invokes another method, a new stack frame is created on top of the current one and a number of object references are passed as parameters to the new frame, making these objects part of the new method’s rootset.

The design of CloudGC emulates this way of organizing root objects by first, maintaining a set of common stack frames and second, creating a separate stack per handled request, each containing its own stack frames. Each stack frame is initialized on creation with a number of object references available from the previous stack frame, which essentially imitates the method invocation procedure. The first stack frame of a request handler is initialized from references of the top common stack frame.

An overview schematic of this design is displayed in Figure 7.1. In CloudGC,
this was implemented as a number of arrays: one static array for holding the
common stack frames and also, one array per request.

7.4.1.2 Graph Roots

Each stack frame constitutes part of the root set of the application; the GC
of the runtime will eventually need to traverse the live objects and they will
be recursively accessible through one of the stack frames that represent the
roots. Thus, each CloudGC stack frame contains a number of references to
objects and in this version, it was implemented by using an array of object
pointers.

7.4.1.3 Graph Objects

Each CloudGC Graph Object is split into two parts: first, a number of object
reference slots; and second, a chunk of non-reference data that represent the
payload of the object. Mimicking the way that objects are created in a
runtime, each CloudGC Graph Object is initialized with a certain number of
object reference slots as well as payload size—similarly to how a class would
have been used as a blueprint. An example part of a CloudGC Object Graph
and a rootset is displayed in Figure 7.2.

Regarding implementation, Java objects were used to represent the Graph
Objects and each of these contain two array pointers: the first points to an
array that stores the references to children-objects; and the second, to a byte
array that acts as the payload of the object.
Figure 7.2: Each CloudGC object maintains a number of pointer slots that can either point to other objects or contain null—represented with 0 in the diagram. Additionally, CloudGC objects contain a byte array for payload, which apart from enlarging the object, it can be used for read and writes that pollute the CPU cache without altering the object graph.

7.4.1.4 Graph Actions

External requests cause a number of randomly selected actions to be executed on the CloudGC object graph with controllable probabilities. For each request, a local stack is created and is initialized with a number of references from the top frame of the common stack. Then, depending on the results of a pseudorandom function (with a controllable seed for repeatability) actions are selected and executed one after the other by the thread handling the request. The available actions are the following:

Add Top Frame: This action is equivalent to a method invocation. A new stack frame is created and added on top of the request’s private stack. The new stack is initialized with a controllable number of references accessible
from the previously top stack frame or, if none available, from the top frame of the shared stack. To select an object for addition, a random walk is performed starting from the roots with controllable probabilities of moving down to the next level.

**Remove Top Frame:** This action is equivalent to a return call from a method. If the top stack frame of the request’s private stack is available, it will be removed. Because all root references stored on this frame are cleared, an immediate result of this action is that a number of objects could immediately become garbage, if their only connection to the object graph was through one of the removed references.

**Allocate Object:** A new object is created with an exponentially randomly selected number of reference slots and payload size. Based on a controllable parameter, the new object is either added to the rootset of the current top frame or an object is randomly selected by performing a random walk starting on the frame’s rootset. All references of the new object are initially set to `null`, i.e. they do not point to any other objects in the beginning. This operation is likely to trigger a GC, if the allocator of the runtime fails to find enough memory for the new object.

**Change Reference:** A random object is selected by performing a random walk starting at the top stack frame. Then with a parameter probability one of its slots is selected and is made to point to a second random object (selected again with a random walk from the top frame) or its reference is cleared by writing `null`. This action is likely to create new garbage by com-
pletely disconnecting a sub graph from the remaining objects; thus, making all objects in this group unreachable from the roots.

**Read Payload:**  This action selects a random object reachable from the top frame and reads some bytes from its payload. This action does not mutate the object graph and is not expected to trigger a GC; however, real applications do handle data and this has locality-of-reference and data-temperature effects.

**Write Payload:**  This action selects a random object reachable from the top frame and updates some bytes from its payload. As before, this action stresses the memory locality patterns of the application and the GC but in a more forceful way.

**Block on Shared Resource:**  This action blocks the thread that handles the request on a shared semaphore, which emulates the effects of accessing an external resource, like a database. Additionally, the semaphore is initialized with a controllable parameter of permits and it blocks for a controllable amount of time. Blocking represents access to external resources the cloud service might be requiring. This results in the system not actively handling the request but its heap is still under pressure because the blocked request’s objects cannot be reclaimed until it has concluded. Essentially, longer waiting on a resource can lead to higher survival rates for objects that would have otherwise died younger.
7.4.2 Most-Objects-Die-Young

Research has shown that the usual memory patterns of applications require objects of two types: first, short-lived objects that usually do not survive more than a handful of GCs; and second, long-lived objects that evade collection and stay on the heap for even as long as the application executes. Essentially, the higher the number of GCs an object has survived, the higher its chance to survive the next GC, a tendency commonly referred to as Most-Objects-Die-Young or the Generational Hypothesis [65].

CloudGC accomplishes this through its usage of stack frames for storing root references: first, objects that make their way into one of the common stack-frame roots will be impossible to be disconnected by a request; and second, objects that are allocated by the request handler on the request’s private stack frames will be very unlikely to survive the execution of the request unless they are selected to be connected to one of the objects originating from the common frames.

7.4.3 Locality of Reference

Memory is organized in various layers: registers; L1, L2, L3, etc. cache; main memory; swap memory; etc. All these are progressively larger, cheaper and slower. Every time a piece of information needs to be processed, it needs to be fetched to the registers from the closest point to the CPU. The best case scenario is that the data is already in a register; it is a bit worse if it is in
the cache; and much worse, if it needs to be read from the main memory. Furthermore, due to prefetching, adjacent data can be brought into the cache before they need to be used. Thus, depending on the hardware and software memory protocols used, references that are accessed one after another that are close in memory addresses will be accessible faster than others (spatial locality). Also, if a reference is used frequently enough, its copy is more likely to exist in a lower-level of the memory hierarchy (temporal locality).

Cloud services, like other applications, primarily handle data. Therefore, their various reading and writing patterns are important due to their significant influence on the speed of memory accesses based on locality. This is also considered by CloudGC, whose read payload and write payload operations are designed to stress the various memory hierarchy protocols. Writes in particular, have an increased chance of interference, particularly in multicore systems. This is because of cache coherence protocols that might invalidate the contents of cache lines and lead to true or false sharing. This phenomenon requires a piece of data to be fetched again from the main memory, which causes significant slowdowns.

### 7.4.4 Data Temperature

Not all objects are accessed equally frequently. More frequently accessed objects are referred to as hot and less frequently accessed as cold. Cold objects are still accessible by the roots and are thus, alive themselves and maintain their descendant objects alive as well. The same effect is achieved
by CloudGC through its stack-frame root design. Objects that are not selected to be propagated to the next stack frame—this includes their graph ancestors—will be unreachable by any subsequent frame; thus, even though they are still alive, they will never be chosen for a graph action. However, objects that make it to the next frame and objects that were very recently allocated and thus likely to be on the top frame, are more likely to be selected for graph actions and will become hot.

Additionally, when fields and slots are selected for accessing, an exponentially distributed random number is selected. Thus, it is significantly more likely to choose fields and slots with smaller indexes than ones with larger. This results in some parts of an object being hotter (in terms of accesses) than others. Also, depending on how frequently reference changes happen, child-objects accessible from smaller-indexed slots are more likely to be selected and are, therefore, hotter as well.

Traversing cold objects during GCs can be an expensive procedure. CloudGC can create cold objects that are connected with hot ones, as is expected in a real application. This way, the runtime is further stressed and potentially will force any relevant optimizations to be triggered. For example, region-based collectors maintain remember sets for all incoming object references per region. If these come from cold objects, then the cold objects do not need to be frequently traversed during GCs, which speeds up the execution time of the GC and minimizes interference.
7.4.5 Reconfigurability

One of the goals of CloudGC is to be highly configurable such that it can cause a large range of impacts on the runtime and its GC. Also, because cloud services patterns can change over time, CloudGC is dynamically reconfigurable, that is, being able to fine-tune its settings without restarting the runtime. Otherwise, the whole heap would need to be rebuilt from scratch and potentially reduce the stress on the dynamic memory management system, something not expected by production cloud services that stay up even for years. CloudGC provides the configuration options for parameters described in Table 7.2, which cover a broad range of its functionality aspects. Also in the table, the values of the settings used for experiments are displayed.

Setting 1 uses the full array of the CloudGC functionality. It contains objects with non-zero payloads and uses a large common stack. Its requests have a certain chance of invoking the blocking action, which forces the thread handling the request to sleep on a semaphore for some period of time. Settings 2 and 3 do not use payloads in their objects; thus, they only contain references to other objects. Setting 2 does not use a shared stack so all of its requests are stateless. Finally, Setting 3 does not use the blocking action, which causes more active requests to be handled at the same time by the runtime.
Table 7.2: The configuration parameters and presets of CloudGC.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Set. 1</th>
<th>Set. 2</th>
<th>Set. 3</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEED</td>
<td>666</td>
<td>666</td>
<td>666</td>
<td>Seed for random object; it is required for reproducibility of the pseudo-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>random actions.</td>
</tr>
<tr>
<td>DEPTH_PROB</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>Likelihood of moving to the next level while performing a random walk.</td>
</tr>
<tr>
<td>MAX_DEPTH</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>Maximum traversal depth before selecting an object during random walks</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>starting from a frame’s rootset.</td>
</tr>
<tr>
<td>MIN_PAYLOAD</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>Parameters for newly created objects’ payload size (exponential).</td>
</tr>
<tr>
<td>MEDIAN_PAYLOAD</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>Parameters for newly created objects’ number of reference slots (exponential).</td>
</tr>
<tr>
<td>MAX_PAYLOAD</td>
<td>1024</td>
<td>0</td>
<td>0</td>
<td>Parameters for newly created objects’ payload size (exponential).</td>
</tr>
<tr>
<td>MIN_REF</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>Parameters for newly created objects’ number of reference slots (exponential).</td>
</tr>
<tr>
<td>MAX_REF</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>Parameters for newly created objects’ number of reference slots (exponential).</td>
</tr>
<tr>
<td>NEW_FRAME_REF</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>Number of references passed on from the previous top frame to the new top</td>
</tr>
<tr>
<td>FRAME_SIZE</td>
<td>100000</td>
<td>100000</td>
<td>100000</td>
<td>Maximum number of references a frame can store.</td>
</tr>
<tr>
<td>INIT_FRAMES</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td>Number of frames to be created on initialization.</td>
</tr>
<tr>
<td>INIT_ALLOC</td>
<td>1000000</td>
<td>0</td>
<td>1000000</td>
<td>Number of allocations per frame on initialization.</td>
</tr>
<tr>
<td>INIT_REF_CHANGES</td>
<td>100</td>
<td>0</td>
<td>100</td>
<td>Number of reference changes per frame on initialization.</td>
</tr>
<tr>
<td>ACTS_PER_REQ</td>
<td>10000</td>
<td>10000</td>
<td>10000</td>
<td>Number of graph actions per request.</td>
</tr>
<tr>
<td>ADD</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>Frequency weights of graph actions that are randomly selected for execution at each request. A particular relative frequency is calculated as: $f_i = \frac{w_i}{\sum w_j}$</td>
</tr>
<tr>
<td>REMOVE</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>ALLOC</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>REFCHANGE</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>READ</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>WRITE</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>BLOCK</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ALLOC_ON_ROOT</td>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
<td>Likelihood a new object will be added directly on the top-frame versus under an object selected through a random walk starting from the top frame.</td>
</tr>
<tr>
<td>BLOCK_SEM_SIZE</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>Number of permits the shared semaphore gives to requests.</td>
</tr>
<tr>
<td>BLOCK_SEM_TIME</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>Time in ms a thread will hold the shared semaphore during a block action.</td>
</tr>
</tbody>
</table>
7.4.6 CloudGC API

The API of CloudGC is based on remote requests that arrive from an external network, following the client/server paradigm. In particular, CloudGC exposes the following HTTP endpoints:

- `/ChangeSettings[?param=value]*`, which updates the values of all passed parameters accordingly and returns a JSON file with all stored parameters after the update.

- `/InitGraph`, which initializes the object graph of the system by (re)creating the common stack and adding new objects to its frames according to the stored parameters.

- `/GraphAction`, which creates a new local stack for handling this request, initializing its first frame with references taken from the top frame of the common stack and then, it executes a number of randomly selected actions according to the parameters.

- `/GetGCStats`, which returns a report regarding recorded GC activity. In the current implementation, the name of the GC component, the times it was invoked and the total time spent executing are displayed using information acquired from GarbageCollectorMXBean.

All these endpoints were mapped to corresponding Java EE Servlets that implemented them.
7.5 GC Policy SLO Satisfaction

A number of experiments were conducted using CloudGC in order to investigate the SLO performance of the four GC policies of the IBM JVM. Since cloud environments do not necessarily provide full isolation among their tenants, the experiments were conducted with and without performance interference and the effect on the co-located tenants was also measured.

7.5.1 Experimental Methodology

Scripts to deploy CloudGC on a local and isolated instance of the PaaS software Cloud Foundry were created. Liberty Buildpack, which uses the IBM J9 JVM for language runtime and the IBM Websphere Liberty Profile application server was used. Load to CloudGC was driven from a computer outside the Cloud Foundry installation, which ran Apache JMeter for varying numbers of parallel clients (1, 8 and 64) and for a total time of 3000s per run. JMeter was set to timeout after 5s, so the maximum recorded response time was 5,000ms. In all cases, a container with memory size of 6G, which resulted in a maximum heap size (Xmx) equal to 4,608MB since the Liberty Buildpack sets the Xmx to 75% of the container’s memory, was used. Additionally, the container had 8 CPU cores available.

A second interfering tenant was setup on the same Cloud Foundry deployment. For the times that it was activated, a second instance of Apache JMeter was firing requests with 8 parallel clients for the same amount of
time. The interfering tenant was written in C++ and was deployed with a C++ buildpack for reasons similar to those explained in Chapter 5.

For each run, the script performed the following:

1. Deleted any previous CloudGC instances and deployed a fresh one using a manifest file. There were four manifest files available, one for each of the four GC policies available in the IBM JVM: Gencon (gen), Balanced (bal), Optavgpause (pau) and Optthruput (thru).

2. Performed an HTTP request to the /ChangeSettings endpoint so that the system’s parameters are initialized to the desired values.

3. Performed an HTTP request to the /InitGraph endpoint so that the shared frames were initialized.

4. If the script was running in interference mode, it invoked JMeter in the background to fire requests to the C++ interfering tenant for 3000s.

5. Made a request to the /GetGCStats endpoint to capture the GC stats before the stress testing.

6. Invoked JMeter, which fired repeated requests to the /GraphActions endpoint for 3000s using either 1, 8, or 64 parallel clients.

7. Made a request to the /GetGCStats endpoint to capture the GC stats after the stress testing.
For each run, JMeter recorded a file with information regarding the requests, including if they were successful or not, their timestamp and their response time. Additionally, the GC stats before and after the stress test were recorded in a separate file. Finally, if the run was an interference run, a second JMeter file with response data was stored regarding the interfering tenant. Finally, regarding the analysis of the response data of CloudGC, the first 3,000 requests were excluded to eliminate any JIT warmup effects. The IBM J9 JIT is set by default to optimize methods that are invoked 3,000 times. The measurements were aggregated per condition and only requests with successful responses were considered. For each condition, the number of responses per response-time in milliseconds was counted and averaged, essentially creating a histogram of responses per response-time (between 1ms and 5,000ms). The cumulative distribution (both absolute and relative) was subsequently calculated from the histogram. Additionally, the GC statistics per run were aggregated by taking the difference of the number of GCs and GC time before and after the stress test and averaging them. The various experimental metrics are defined and explained in Table 7.3. The three different settings of CloudGC were also experimentally evaluated using the aforementioned methodology. The experimental results, which are omitted for brevity, suggest that the benchmark has a large range of stressing the runtime and GC in a variety of ways through its configurations. In particular, Setting 1 was affected the most as more parallel clients were added. Additionally, even though Setting 3 started as the fastest when fewer clients
Table 7.3: List of dependent variables and experimental metrics captured while experimenting with *CloudGC*.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Histogram</td>
<td>For every millisecond between 1 and 5,000, count the number of requests that had a response time equal to this millisecond.</td>
<td>( H(t) =</td>
</tr>
<tr>
<td>Cumulative Mass</td>
<td>For every millisecond between 1 and 5,000, count the number of requests that had a response time less than or equal to this millisecond.</td>
<td>( F(t + 1) = H(t + 1) + D(t) )</td>
</tr>
<tr>
<td>Throughput</td>
<td>Average number of requests served per second.</td>
<td>( \text{thru} = \frac{F(5000)}{3000} )</td>
</tr>
<tr>
<td>Rel StdDev of Resp Time</td>
<td>Percentage of the variance of the response times over the average.</td>
<td>( \text{relDevResp} = \frac{\text{stdDevResp}}{\text{avgResp}} )</td>
</tr>
<tr>
<td>GC Time per GC</td>
<td>Average time each GC event took to complete in milliseconds.</td>
<td>( \text{GCTime}/\text{GC} = \frac{\text{GCTime}}{\text{GCs}} )</td>
</tr>
<tr>
<td>GCs per second</td>
<td>Number of GC events performed per second.</td>
<td>( \text{GCs/sec} = \frac{\text{GCs}}{3000} )</td>
</tr>
<tr>
<td>GC Time per second</td>
<td>Average time the GC was running per second in milliseconds.</td>
<td>( \text{GCTime/sec} = \frac{\text{GCTime}}{3000} )</td>
</tr>
<tr>
<td>GCs per request</td>
<td>Number of GC events performed per request.</td>
<td>( \text{GCs/req} = \frac{\text{GCs}}{F(5000)} )</td>
</tr>
<tr>
<td>GC Time per request</td>
<td>Average time the GC was running per request in milliseconds.</td>
<td>( \text{GCTime/sec} = \frac{\text{GCTime}}{F(5000)} )</td>
</tr>
<tr>
<td>Throughput Reduction</td>
<td>Measures throughput performance interference as a percentage of the reductions over a baseline non-interference throughput. The higher this number, the higher the performance interference.</td>
<td>( \text{thruRed} = \frac{\text{thru} - \text{noInfThru}}{\text{noInfThru}} )</td>
</tr>
<tr>
<td>Response Time Increase</td>
<td>Measures response time performance interference as a percentage of the increases over a baseline non-interference response time. The higher this number, the higher the performance interference.</td>
<td>( \text{respInc} = \frac{\text{resp} - \text{noInfResp}}{\text{noInfResp}} )</td>
</tr>
</tbody>
</table>
were used, it became slower than Setting 2 as more clients were added, which should be attributed to the stateless design Setting 2 implements. Furthermore, unlike the other two settings, the response times of Setting 3 became more stable as more clients were added, presumably due to the added contention on the shared resource it utilizes. Finally, since Setting 2 lacked an object-graph initialization phase, the runtime did not have the opportunity to perform enough heap expansions in the beginning, a warm up cost that was subsequently transferred to a portion of its requests.

7.5.2 Foreground Response-Time

The various experimentally acquired cumulative mass results are displayed in the graphs of Figure 7.3. These graphs do not show percentages but the actual number of requests completed within a certain time threshold. These numbers showcase the differences between throughput and on-time throughput that the four GC policies possess. Since there were no comparisons across different numbers of clients and settings, the graphs are not presented in the same scale.

7.5.2.1 No interference

In all nine non-interference cases, Gencon, which is the default GC policy, eventually ended up on top of the other three policies. This means that Gencon had the overall best throughput. Additionally, Gencon outperformed the other policies in terms of on-time throughput regarding looser SLO time
Figure 7.3: Graphs displaying the cumulative mass of response times. The x-axes describe the response time in ms; whereas the y-axes denote the cumulative mass of the response time in thousands of samples. The top nine show the non-interference experiments, the bottom nine, the ones with interference enabled.
limits for all but one case. Even though Optthruput, the parallel mark-sweep-compact collector, was able to respond faster than Gencon for the majority of the requests, it was not able to complete as many.

Balanced, the region-based collector performed second-best in on-time throughput for 5 out of 9 cases, none of which was for Setting 3. Instead, Balanced did well for higher loads in Setting 1 and lower loads in Setting 2. Setting 2 is close to a best-case scenario for Balanced in theory: since its requests do not use the common stack, they can essentially function independently from each other, especially when placed in separate regions. Setting 1 uses semaphores and the requests are occasionally blocked on them, which gave the Balanced threads the time to perform their more expensive maintenance activity (managing multiple regions can be costly because extra data, such as remember sets, is required). However, because Setting 3 does not block the threads, extra contention arose between the GC threads and the mutator threads, leading into reduced performance.

An interesting frequency distribution was measured for Optthruput with Setting 3 and 64 clients. The results indicate that the responses were grouped into two normally distributed parts. This can be explained by the increased contention between GC threads and mutator threads. The GC threads had to traverse the whole object graph (including any old common objects, since Optthruput is not generational) and the mutator threads had to work non-stop as no blocking actions take place in Setting 3. The effect is that when there was enough heap space, requests were served fast; whereas when the

151
system ran out of space, stop-the-world GCs significantly impeded the re-
response time of any requests served at that time.

*Optavgpause* is a GC policy similar to *Optthruput* but can perform concur-
rent marking to minimize the length of the stop-the-world phases. The best ex-
ample of this GC policy performing as expected can be seen in Setting 3 for 64 clients. Unlike *Optthruput*, whose requests were served within two dis-
cinct groups of response times, *Optavgpause* minimized this gap and as a re-
result it provided improved response-time limits for tight SLO thresholds.

**7.5.2.2 With interference**

Unlike the non-interference experiments, *Gencon* was not always the best policy regarding on-time throughput but it was still the best in the majority of the conditions, including all 64-client cases. Overall, *Balanced* performed second best; *Optthruput* and *Optavgpause* seem to be in a tie for third place as their results were mixed across the various conditions.

The response-time frequency results measured for Setting 1 were the most similar across the various GC policies. For 1 client in particular, the four curves were virtually indistinguishable but gradually diverged as more clients were added.

Setting 2, the stateless setting of *CloudGC*, produced an interesting outcome for the *Balanced* GC policy. Although *Balanced* started off at the top with lower thread counts, it gradually became worse than *Gencon* as more clients are added. For 8 clients in particular, *Balanced* also clearly outperformed
Gencon for this condition in terms of consistent response times. This can be seen on the graph by the sharper slope that Balanced registered.

Setting 3 again outlines the problems of the more traditional Optavgpause and Optthruput GC policies as they either have a distinct plateau on their frequency plots or do not rise as fast as the curves of the other two policies. Additionally, when in comparison to the non-interference case, Gencon and Balanced were more resilient in handling the reduced-resource environment. Setting 2 with 8 clients illustrates this the best: when interference was introduced, the graphs of Optthruput and Optavgpause both dropped lower than the other two policies and also, did not rise as fast. Overall, there were not many differences in the general shapes of the curves and the ordering of the GC policies. This means that the two results support each other and the general conclusions drawn from the previous section also apply when interference is added.

7.5.3 GC Statistics

The average number of GCs per request and average execution time per GC are displayed in Figure 7.4 and Figure 7.5 respectively.

7.5.3.1 No interference

In 7 out of 9 cases, Gencon performed the most GCs per request. Gencon utilizes a nursery, which constitutes only a small portion of the heap, for allocating young objects. Because the nursery is so small, it filled up
Figure 7.4: Experimentally measured number of GCs per request. The top nine show the non-interference experiments, the bottom nine, the ones with interference enabled.
Figure 7.5: Experimentally measured number of GC time per request. The top nine show the non-interference experiments, the bottom nine, the ones with interference enabled.
faster, which led to many more copying GC events, although nearly zero mark-sweep-compact GC events in the tenure space. When paired up with the response results of Figure 7.3, *Gencon* in general completed more on-time requests, while conducting more GCs per request. Thus, copying young objects back and forth before promoting them to tenure and essentially forgetting about them was quite successful for *CloudGC*.

Regarding Settings 1 and 3, the number of GCs was as follows: The most were by *Gencon*, the second most by *Balanced*, the third most by *Optavgpause*, and the least by *Optthruput*. This is compatible with the policies’ design: *Optthruput* is aiming to minimize the number of GCs and is perfectly willing to potentially penalize only a small subset of the requests with long stop-the-world phases. *Gencon* and *Balanced* use nursery spaces, which require frequent copying collections. *Optavgpause* is almost like *Optthruput* but because it performs some concurrent marking, it can interfere with the response time of some requests, prolonging it long enough such that a few are delayed by its stop-the-world mark-sweep phase.

For Setting 2, however, the number of GCs per request, especially for higher loads, reversed. Because Setting 2 handles its requests in a stateless way, it is possible that the JIT performed some optimization, such as stack allocation. Because such a task would require coordination between both the GC and JIT systems, it is conceivable that this task has only been implemented for the default, *Gencon* policy. This would explain why Setting 2 is the only one that the relative number of GCs per request changed so dramatically as the
load increased but only for Gencon. Additionally, the response-time results of Figure 7.3, indicate that Gencon was the best performing GC policy and with the largest recorded margin among our experiments. Moreover, the concurrent marking performed by Optavgpause was more successful in terms of number of GCs per request than Optthruput. This was the case for Setting 2 again because of its statelessness. In particular, objects in this Setting cannot escape their request and thus, their allocating thread. This reduces the contention among threads while performing concurrent marking.

Furthermore, the GC time generally correlates with the number of GCs. Intuitively, the more GC events that happen per request, the more time it will take for them to complete. However, Gencon registered different results regarding this matter. Although Gencon generally had the most GCs per request, it also had the lowest GC time per request. This indicates that the extra copying GC events that Gencon performs in its nursery are significantly less time consuming than GC events of other policies. Balanced was able to outperform Gencon in Setting 2 but only for lower client counts. Again for Setting 2 but for higher client counts, Optavgpause needed less GC time per request than Optthruput. However, Optavgpause in general scored much higher than the other methods, especially for lower numbers of clients. This is expected because Optavgpause is running concurrent marking while the requests are executing; when there is less pressure on the system, more GC threads are available to perform marking.
7.5.3.2 With interference

The number of GCs per request was reduced in comparison to the non-interference experiments but the opposite happened for the GC time per request. In other words, performance interference caused fewer GCs that required more time to complete proportionally to the average response time. From this, we can infer that GC threads were starved in terms of CPU resources.

In general, the relative differences between the four GC policies remained unchanged for both our GC-request metrics. Thus, the main points from the discussion in the previous non-interference section still stand.

Nevertheless, some differences were measured for Setting 2, the stateless and thus, short-lived-objects CloudGC setting. In particular, Optavgpause executed more GCs than Optthruput for lower client clouds but still fewer, for 64. This implies that its concurrent marking threads have lower priority and were largely starved and unable to perform their duties as well as in the non-interference mode. Moreover, Gencon performed fewer GCs per request than Optavgpause and Optthruput for all client counts and Balanced for higher numbers of clients. Since the GC threads of Gencon have to stop the world to conduct collection of the nursery space, they were less susceptible to starvation as they were only competing against the background C++ tenant and not the foreground mutator as in the Balanced and Optavgpause cases.

All in all, first Gencon and secondarily Balanced were better able to resist
the negative effect of an aggressive co-located tenant in this cloud and containerized deployment.

7.5.4 Background C++ Tenant

In a multitenant cloud, it is also of interest to minimize the performance interference induced by a co-located tenant. Cloud providers can place tenants more densely while still satisfying their SLOs; cloud clients also benefit especially when the various co-located tenants belong to the same company or even micro-services oriented application. Thus, the remainder of this section explores which GC policy resulted in the smallest interference to the background C++ tenant in terms of throughput and response time.

First, Figure 7.6 displays the relative throughput reductions of the background tenant. Overall, the more CloudGC clients were added, the higher was the measured interference. Naturally, in a multitenant cloud system lacking hard resource limitations, the higher the activity of the other co-tenants, the lower the slice of resources available, which leads to QoS degradation.

Regarding comparing the four GC policies, Gencon induced the lowest throughput interference in 5 out of 9 cases, followed by Balanced with the lowest in 4. Therefore, not only were Optthruput and Optavgpause not as effective in supporting their own tenant, they also interfered more with the neighboring tenant. This can be explained since Optavgpause and Optthruput perform a lot of work in vain, scanning again and again old objects that are unlikely to be garbage.
Second, Figure 7.7 displays the relative response-time increases. In the 1-client case for Settings 1 and 2, the interference was near zero but not for Setting 3. Setting 3 had both a large common stack and no blocking actions; thus, it needed to consume significant resources even for the low-load 1-client case. In contrast, Setting 1 interfered very little even for 64 clients; Setting 2 interfered a lot only for 64 clients. This result outlines—in retrospect—another ability of the CloudGC settings, being able to induce response-time interference in various levels even for the same number of clients.
In all cases, response-time interference was more like an on-off switch unlike throughput interference, which was more progressive. Additionally, the variation of response-time interference was much higher than throughput interference as the former ranged from near 0% to as high as 190%, whereas the latter from a minimum of 15% to a maximum of 65%. This further illustrates the effects of performance interference for systems that are sensitive to response-time related SLO violations. This suggests that as more performance interference is introduced, the system is at a higher risk of a sudden
QoS drop and thus, a response-time SLO violation. Comparing GC policies, *Gencon* induced the least interference in the majority of the conditions, followed by *Balanced*. *Optavgpause* and *Optthruput* largely caused the highest response-time interference delays.

### 7.5.5 GC Policies Discussion

The experiments resulted in the following observations and conclusions comparing the performance of *CloudGC* using the four available GC policies of the IBM J9 Java Runtime:

*Gencon*, the default GC policy, was in the majority of cases the best performing GC policy regarding both SLOs of on-time throughput and on-time responding for various thresholds. *Gencon*, most of the time, performed the highest number of GCs per request but simultaneously had the lowest GC time per request. *Gencon* achieves this by conducting very frequent copying GCs on its young objects located on the nursery space. Objects that have aged sufficiently are moved to the tenure space, which is rarely collected. Furthermore, significantly different measurements between this and the other policies for the stateless Setting 2 of *CloudGC* could be explained by the JIT aggressively allocating objects on the stack only for this GC policy.

*Balanced*, the region-based collector, performed second best in terms of response SLOs for the majority of the experimental conditions. For the two *CloudGC* settings that were not stateless and did not use long-lived objects, it conducted the second highest number of GCs per request and its GC
threads ran for the second highest time per request. However, regarding the stateless Setting 2 and with fewer parallel clients, it had the lowest number of GCs per request and the lowest GC time per request. For this setting, it also performed significantly better than Optthruput and Optavgpause. Finally, in all conditions, Balanced did not have the significant response-time variations that were measured for Optthruput and Optavgpause.

Optthruput, the parallel mark-sweep-compact collector, never produced the highest throughput, as its title implies. Gencon measured consistently higher throughput than Optthruput across all testing conditions. Optthruput had the lowest number of GCs for the two non-stateless CloudGC settings but the highest for the stateless one, especially for higher client numbers. However, the very few GCs resulted in its cumulative mass of response-time creating visible plateaus as some requests were served with large delays, which best matched the predictions of our theoretical model. It also had the highest GC time per request for these conditions and overall, more GC time per request than Gencon. The drawbacks of Optthruput were particularly prominent in the stateless Setting 2. This is expected because this setting does not have any long-lived objects and because Optthruput does not differentiate between old and young objects. Therefore, every time a GC took place, all of the objects and the heap needed to be scanned for garbage.

Optavgpause, the concurrent-marking version of Optthruput, performed similarly to Optthruput. Most of the time, it scored the highest GC time per request especially for lower loads. This is expected since its threads are
running at the same time as the mutator threads are serving the requests. When interference was added, the response-time frequency graphs were comparable to the non-interference tests. Gencon and Balanced were able to better mitigate performance interference over Optthruput and Optavgpause. However, Gencon was not always the best SLO satisfying policy but still best in the majority of cases. Balanced performed the best in some conditions, which showcases that an ideal circumstance for selecting Balanced over the default Gencon is when the application is expected to have low load and virtually no long-lived or escaping objects in conjunction with performance interference present from a co-located tenant. Furthermore, fewer GCs per request, but with more time per request were measured for all GC policies. Therefore, interference caused GCs to occur less frequently, but last proportionally longer. This could be attributed to resource starvation.

Finally, as the load of a tenant increased, so did the performance interference it induced in its co-located tenants. Throughput interference was less intense and occurred later that response-time interference as the load increased. Systems that are sensitive to response-time SLO violations can experience more severe and instantaneous suffering from increased activity of an interfering tenant. First Gencon and then Balanced induced the least performance interference to the background C++ tenant. Optavgpause and Optthruput apparently wasted time and resources on repeatedly and unnecessarily scanning the same old and long-lived objects. Thus, the “most objects die young” hypothesis is also relevant for reducing performance interference.
7.6 Conclusion

With the advent of cloud computing, the inner workings of high-level language runtimes, such as GC cannot be neglected as they can have a significant impact on the system’s QoS and thus, its ability to satisfy its SLOs. Additionally, multitenant clouds with soft resource limits allow tenants to cause performance interference with other co-located tenants, which can in turn cause QoS to degrade and lead to SLO violations and even, financial penalties.

Focusing on GC and its potentially deleterious effects on SLO satisfaction, CloudGC, a GC-oriented stressing benchmark for PaaS clouds was proposed. CloudGC was then used to experimentally evaluate the SLO performance of the four GC policies that are provided in the IBM JVM.

Crucial findings include: First, the default GC policy, Gencon, is the most performant in the majority of the experimental conditions. Second, Balanced, the region-based collector, was overall second best but it was the best with the CloudGC setting that used only short-lived and non-escaping objects with lower client loads. Third, the two traditional mark-and-sweep GC policies, Optavgpause and Optthruput, in general did not perform on par with the other two policies, mostly because of their inability to take advantage of the most-objects-die-young hypothesis, which is highly relevant in cloud computing and CloudGC. Fourth, Gencon and Balanced were better able to mitigate the effects of performance interference and also induced the least
interference themselves. Fifth, the various settings of CloudGC were able to induce performance interference in various levels even for the same number of clients. Finally, although throughput interference increased progressively as the load increased, response-time interference was more abrupt and occurred in much higher rates. Thus, systems whose SLOs are tied to response times are more vulnerable to performance interference as they risk violating their SLOs abruptly in the presence of increased performance interference.

In the future, this work can be expanded in a number of ways: A more thorough analysis of the parameters of CloudGC could be performed by experimenting with small variations of only one parameter at a time. This, can also lead to new CloudGC preset settings. Additionally, CloudGC could be used to compare the performance of various Cloud platform technologies, cloud providers, GC policies and runtimes. Finally, the experimental results for each run could be analyzed and aggregated to a smaller set of SLO-aware benchmark scores, which could standardize the comparison procedure.
As this work has already established, benefits of multitenancy on the cloud come at the price of performance interference, which can result in unpredictable behavior and SLO violations. Depending on the Garbage Collection (GC) policy a cloud application uses, it might require CPU utilization spikes, which are caused by the GC threads that clean up the heap. However, if the application is running in low load, it might not reclaim memory as soon as possible, especially since this can happen at the expense of interfering with any co-located tenants. In this chapter, a technique of GC elasticity is proposed that scales down the GC threads and heap size at times of low load, while running a preemptive GC. Experimental results suggest that Elastic GC reduces SLO violations on co-located tenants, if the GC-scaled tenant maintains a large object graph and selects a GC policy that utilizes long stop-the-world phases. Elements of this chapter are scheduled to be pub-
lished in the Proceedings of IEEE SASO 2018 [96] and resulted in a planned patent application with IBM.

8.1 Motivation

The load of cloud applications varies, sometimes even with predictable and periodic patterns. Clouds scale their applications by adding/removing instances or increasing/decreasing the resources that are awarded to them in order to cope with these load changes. Thus, scaling normally happens at the process level. However, the resources consumed by specific tasks of an instance are not individually scaled, even if doing so would reduce its performance interference on co-located tenants without violating its SLOs.

This issue is more prominent in background, CPU-intensive computations that are performed by high-level language runtimes, such as GC. Certain GC policies require lengthy stop-the-world phases that use all the available computing resources to scan all of the heap for dead objects. Nevertheless, if the application is currently experiencing low load, there is no need to use that much CPU at the expense of performance interference.

In addition, large memory heaps can increase the time it takes for GC threads to scan them. In particular, the worst-case complexity for a collection is proportional to the heap size. However, due to the stateless design of cloud client/server applications, a large portion of the heap is required to maintain data for requests only as long as they are active. Consequently, during times
of low load, an application should not require as much heap to successfully satisfy its requests; maintaining a larger-than-required heap can cause even longer GC CPU spikes and increased performance interference.

8.2 Related Work

Scaling in PaaS systems has been facilitated with triggers activated when particular resource consumptions of an instance reach certain thresholds. In particular, services that automatically perform this kind of scaling, either to the tenant application as a whole [116] or to its most stressed components [66], have been proposed.

At the PaaS level, Blagodurov et al. proposed using Linux cgroups with CPU weights to enforce priorities on foreground versus background tasks running on the same host [33]. Nevertheless, a more fine-grained scaling solution that controls parts of a runtime might benefit SLO enforcement.

At the IaaS level, Akshtat et al. studied the workload traces of a real datacenter and found that placing instances so that concurrent utilization peaks are avoided leads to improved power consumption as well as reduced SLA violations in terms of available CPU% [130]. Additionally, Ripal et al. proposed using CPU% limits and allocating extra resources when available to mitigate performance interference of co-located VMs and enable acceptable QoS [85]. However, using CPU limits means that if an instance is in low load, its CPU portion will not be used at all and part of the CPU resource
will be idle, which is not efficient and thus, only used in dedicated clouds. Furthermore, Steinder et al. proposed a series of VM scheduling optimizations for IaaS systems, including pinning virtualized CPUs to certain physical CPUs as a means to improve performance [123]. Another IaaS work by Rodero et al., proposed pinning VMs to specific physical cores in order to keep the temperature of each core as low as possible, essentially aiming to minimize energy consumption without degrading the tenants performance [109]. On the hardware level: Chengxiang proposed PIN-Cache, a CPU cache management system that reduces performance interference among processes running on the same physical host [118].

At the SaaS level, Krebs et al. proposed a request-scheduling algorithm [74] that throttles the handling of requests for tenants that use more than their allocated share of resources. PaaS applications, however, normally do not handle multiple tenants on a single instance; thus, alternative types of intra-process resource management are required.

Relevant work has also been conducted inside the runtimes and focusing on the GC: Soman et al. studied dynamic memory management inside multi-tenant runtimes—that is, runtimes that can run more than one application at the same time and share the OS process [120]. For this work, the authors focused in mitigating the performance interference GC caused by one tenant has on the others and did so by maintaining per-tenant young generations. Additionally, Balanced GC [3] from IBM is a policy that aims to minimize GC pause times. Balanced GC splits the heap into various regions, which
can be marked and collected independently, potentially allowing the mutator threads to continue uninterrupted for longer time periods. Neither of these techniques take load, SLO violations or multitenancy into consideration.

8.3 Elastic GC

Elastic GC is proposed to mitigate the problem of GC interference on clouds. The goal is to flatten GC-related spikes in resource consumption when the tenant is easily satisfying its SLOs during periods of low load. Thus, any unused resources will be made available to co-located tenants, which in turn can help them improve their SLO satisfaction during high loads.

8.3.1 Design and Implementation

The design of Elastic GC includes the following tasks: First, monitoring the load and/or SLO satisfaction of a tenant with a daemon thread that runs periodically; second, deciding and enforcing an appropriate heap size; third, deciding and enforcing an appropriate number of CPU cores (or threads) the GC is allowed to use; and fourth, explicitly invoking the GC at periods of low load. CPU limits are enforced through thread pinning; however, other techniques such as dynamically changing the thread pool sizes could have been used instead. The proposed technique is outlined in the flowchart of Figure 8.1.

A prototype of Elastic GC was implemented for the IBM JVM and the
Figure 8.1: Overall algorithm of the proposed Elastic GC technique that scales the runtime’s resources aiming to minimize interference on co-located tenants.

Liberty Java EE application server. In particular, a daemon thread was used that ran periodically and collected statistics on the current levels of load the tenant is experiencing through the network files of the container. When the load became lower than a user-defined threshold, the heap was reduced to 80% of its maximum size and the GC threads were pinned to a single CPU core (two hardware threads). When the load became higher than the threshold, the heap size was reset to the maximum and the GC threads were unpinned and allowed to use all CPUs.
Changing the heap size requires an API exposed by the runtime (for example, the IBM JVM exposes softmx for this purpose). Throttling the GC threads can be done in two ways: first, by explicitly changing the number of GC threads, if the runtime provides such an API, and second, by pinning the GC threads on specific CPU cores. Pinning was chosen as it does not require such an API and also, has the added benefit of forcing the OS to place all GC threads of all low-load tenants on the same cores. Therefore, pinning always leaves the remaining cores free to be used by other co-located tenants with reduced interference. Finally, for the heap-size change to take effect in the IBM JVM, a GC needs to be triggered. Therefore, GC threads were first restricted; second, the heap was resized; and finally, a GC was invoked.

Apart from benefits, Elastic GC comes with the drawback of increasing the frequency of collections. Since the heap size is reduced, the memory will be used up faster; thus, the GC will need to run more often. Consequently, it is expected that Elastic GC, while in low-load mode, will cause more frequent but less CPU-intensive collections. It should depend on the characteristics of the application and its SLO targets, whether Elastic GC is a suitable technique or not.

### 8.4 Experimental Evaluation

The Elastic GC algorithm was deployed on top of the Cloud Foundry Liberty buildpack that includes the IBM JVM and Liberty Profile. CloudGC was
used for the tenant code as it produces a predictable and controllable GC load. The application was deployed on a local and isolated installation of Cloud Foundry. Additionally, a C++ application server tenant was used as a background tenant and was also deployed on the same cloud. Separate instances of JMeter were deployed on a machine outside the cloud and on different hardware to drive the load. Only a single parallel request was used for the CloudGC Java tenant so that it runs in low-load mode; whereas, 16 parallel requests were being fired to the C++ tenant.

The tests were repeated multiple times for each combination of the three CloudGC settings, the four IBM JVM GC policies—which are Gencon (gen), Balanced (bal), Optavgpause (pau) and Optthroughput (thr)—and with the Elastic GC mode on and off. Each of these tests lasted for 1000s, such that there was enough time for multiple GC events to take place. Per-second throughput and response-time data were collected from both the Java and the C++ tenants.

The performance of the C++ tenant while it ran without interference was used as a baseline. Various levels of throughput and response time SLO satisfaction were calculated relative to this baseline. Afterwards, the number of seconds that an SLO violation occurred was extracted and the absolute differences between Elastic GC and the default mode were plotted in the graphs of Figure 8.2. The aggregated response time overhead of the Java EE CloudGC tenant is displayed in Figure 8.3.
8.4.1 Discussion

The experimental results suggest that Elastic GC had a negative effect in satisfying tight SLOs of Setting 1 using Optthroughput and Optavgpause as well as tight SLOs of Setting 3 using any GC policy but Gencon. The SLO satisfaction of Setting 2 remained largely unaffected. Nevertheless, gains in SLO satisfaction were recorded for Setting 3, using Optthroughput and for looser
Figure 8.3: Experimentally acquired overhead measurements of the Elastic GC algorithm on the foreground Java EE tenant.

SLO targets. Overall, the throughput and response time results were similar: this is expected because the C++ tenant was receiving repeated requests and any throughput drop was caused by an increased response time.

Regarding the settings of CloudGC, unlike 1 and 3, Setting 2 does not maintain a sizable and persistent object graph. Thus, the GC is not expected to require long bursts of CPU activity to scan the heap, which would otherwise cause performance interference; it is reasonable that no changes in SLO satisfaction were measured between the default and Elastic GC modes.

CloudGC Settings 1 and 3 differ in the following two ways: First, Setting 1 maintains objects with non-zero payloads; and second, Setting 3 does not use the blocking action. However, since CloudGC was run in low load mode, only one request was fired at it in parallel; thus, the only difference that must have had an effect on this experiment, is the payload size. Essentially, objects of Setting 3 were more packed with references to other objects and the GC threads must have performed even more work in order to complete their task. Furthermore, as discussed earlier, Elastic GC is expected to cause
more frequent GCs. These GC events will cause more frequent but lower-intensity interference; consequently, it is reasonable that the satisfaction of tight SLOs decreased for both these settings. Nevertheless, regarding looser SLOs and as long as the GC performs a significant amount of concentrated work, benefits were expected. This combination of events took place for Setting 3 using Optthroughput.

Concerning the differences between GC policies, Optthroughput is the best candidate for benefiting from a technique such as Elastic GC because it is expected to cause the largest GC spikes. Both Gencon and Balanced are generational and only scan a fraction of the heap at a time. In addition, even though Optavgpause performs full heap scans, it also amortizes the marking cost across time by executing marking concurrently with the mutator threads. Finally, the overhead results on the Java tenant suggest that EGC caused the response time increase around 50% for Setting 3 and around 10% for Setting 1, using Optavgpause or Optthroughput. This offers further evidence that the Java EE tenant was essentially throttled by the EGC technique. In conjunction with the very large error bars, the response time was longer in these cases because a request was delayed by a GC cycle that ran for a longer time, as the GC threads handling it were pinned down.

8.4.2 Resource Consumption Investigation

Elastic GC was beneficial for only one combination of GC policy and type of application. To further investigate the reasons of this phenomenon, a
Figure 8.4: Experimentally acquired C++ tenant’s throughput (left) and response time (right) measurements of a sample run to illustrate the differences caused by Elastic GC over the default.

The second test was run with Setting 3 and Optthroughput. The CPU utilization and Resident Set Size (RSS) of the Java tenant was also enabled for these experiments. Profiling was not included in the main experiments, as it would have incurred an extra overhead, skewing the results.

Because collections happen at unpredictable times, the time-series results could not be just aggregated per-second. This would have caused any GC-related spikes of one run to be aggregated with no-GC activity times of the other runs. Thus, the raw results of a sample and indicative run are first displayed in Figure 8.4 and Figure 8.5. These results indicate that the default mode caused higher CPU% GC spikes and Elastic GC caused more frequent GCs as expected. Furthermore, the performance of the foreground C++ tenant was affected less with Elastic GC.

The measurements of the multiple runs were aggregated by comparing the performance of the foreground tenant during periods of significant interference, which was defined as samples one standard deviation away from the median. The aggregate results of the foreground C++ tenant are reported
Figure 8.5: Experimentally acquired CPU utilization measurements of one sample run of the foreground Java EE tenant reveal that Elastic GC caps the spikes during periods of low loads in comparison to the default.

Figure 8.6: Experimentally acquired aggregate throughput (left) and response time (right) of the C++ tenant during periods of interference.

in Figure 8.6 and those of the background Java tenant in Figure 8.7.

The experiments suggest that Elastic GC improved the average throughput of the foreground tenant 1.49 times and decreased its average response time by a factor of 0.53 during interference periods. Additionally, Elastic GC decreased the CPU utilization of the background tenant by 0.30 during GC spikes. Furthermore, the resident memory of the background tenant was reduced by 0.83 for Elastic GC. This ratio is close to the maximum expected 4 over 5 (0.8), since this is how much the heap size was reduced in these tests. Overall, these results highlight the reasons Elastic GC can reduce perfor-
Figure 8.7: Experimentally acquired aggregate CPU utilization and resident memory set size of the Java EE tenant.

8.5 Conclusions

Packing applications together through multitenancy can improve the efficiency of cloud providers and reduce costs. However, tenants can interfere with each other, particularly through contention of any shared resources, such as CPU. Sudden CPU spikes, in particular, can even cause SLO violations on co-located tenants. This is acceptable during periods of high load as a tenant is trying to satisfy its requests in the best possible way. However, when a tenant receives low loads, it might not require completion of its tasks as quickly as possible. This is more evident for background operations performed by language runtimes, such as GC.

To mitigate the problem of increased performance interference caused by the GC, the technique of Elastic GC was proposed and evaluated. During low loads, Elastic GC reduces heap size and restricts GC threads to a subset of the available CPU cores. This results in more frequent but less CPU-intensive
Elastic GC was evaluated on the PaaS Cloud Foundry software using the Liberty buildpack, which contains the IBM JVM and the Liberty Profile application server. The three default settings of CloudGC were used as benchmarks; the results suggested that Elastic GC improved SLO satisfaction for one of the settings but only when the GC policy Optthroughput was used. This was the case for two reasons: first, this setting causes the highest GC activity and second, this GC policy causes the longest and most concentrated GC utilization spikes. Further investigation on this combination revealed that CPU spikes were indeed amortized and the resident size was decreased to the expected levels.

Thus, Elastic GC can help mitigate the performance interference of low-load tenants on other co-located tenants in multitenant clouds. However, this can happen only if the application performs lengthy and CPU-intensive collections, which depends on both its execution patterns as well as the selected GC policy.

Apart from scaling down the GC and heap size, the same idea could be also applied on other resource-intensive components of language runtimes. For instance, class loading and Just-in-Time compiling are both tasks that are executed in parallel and can cause unnecessary CPU utilization spikes. Future work can look into the potential benefits of restricting their resource usage at times of low load, such that performance interference is reduced.

Finally, ideas elaborated in this chapter go against the mainstream tendencies...
of using as many resources as possible to solve a problem in the fastest time possible; instead, there are times that throttling down a part might be more beneficial for the whole.
Chapter 9

Conclusions

Web and mobile applications execute only a part of their computations locally; instead, the majority of the work is outsourced to a remotely running, application server. Cloud computing abstracts resources and offers them transparently over a network—most commonly, the Internet. Language runtimes, which support high-level languages that are predominantly chosen for implementing application servers, are deployed on the cloud.

In this work, a number of topics at the intersection of clouds and runtimes were explored, such as multitenancy, scaling, performance interference and satisfaction of non-functional requirements. To this end, various mathematical models, optimization techniques, cloud/runtime methodologies and benchmarking tools were developed and evaluated. Elements of this thesis have been disseminated in various academic venues and have also been disclosed in patent applications.
Initially, a novel cloud multitenancy model was proposed and a prototype of its main component was implemented and evaluated. Application Server as a Service (ASaaS) enables the full and secure sharing of the hardware/software stack among tenants that run on the same multitenant application server. This solution is better suited for small applications, such as microservices. The experimental evaluation conducted revealed memory gains, linear to the number of tenants packed on the same multitenant server.

Nevertheless, multitenancy comes at price, that of performance interference. This phenomenon was investigated for containerized platform clouds: a theoretical model for the expected CPU utilization of tenants running and scaling on the same host was devised and evaluated. In addition, a method of quantifying the problem of performance interference was proposed, that of measuring how much a tenant slows down in the presence of an aggressive tenant that hijacks a specific resource. Such a set of Cloud Burner tenants was developed and this methodology was then used to profile the resource slowdown patterns of a number of applications. Performance interference and resource slowdown insights gained from this work, largely motivated the remainder of the conducted research.

Scaling is used to handle increasing loads of cloud applications. However, any experience gained from a first run is not passed on to subsequent instances on platform clouds. This results in startup slowdowns as the new language runtime instances need to recreate a number of dynamic artifacts before they are ready to optimally serve clients. Furthermore, constructing
these dynamically compiled artifacts is a highly CPU-intensive procedure; thus, warming up these scaled instances increases performance interference with any co-located tenants. To alleviate this, the platform cloud technique of Dynamically Compiled Artifact Sharing (DCAS) was proposed and evaluated. DCAS uses a cloud service that securely stores and versions artifacts of subscribing applications. Safely circumventing limitations, these artifacts are packed into the application’s image and are thus, immediately available to new, scaled instances. Experimental results suggested significant reductions in startup time and error rates as well as reduced slowdowns on co-located tenants.

The client/server architecture, commonly implemented by cloud applications, conducts work in a request-based manner. These requests normally require small amounts of computation and need to respond within seconds. Cloud providers and their users formally agree on what constitutes an acceptable response time (as well as other non-functional requirements) with thresholds that are called Service Level Objectives (SLOs). To better understand how SLO satisfaction works for a certain number of connected clients that an application server needs to satisfy, a mathematical model was devised and evaluated. Afterwards, using this model, a constant-time theoretical calculation of the ideal number of scaled instances required to satisfy the SLOs was evaluated. In addition, the insights of the mathematical model led to an application-server optimization technique that reorders the execution of requests, such that SLO satisfaction is maximized.
Language runtimes offer transparent memory reclamation through Garbage Collection (GC). Nevertheless, GC is a CPU-intensive task that occasionally requires the application threads to block and wait. This is a twofold problem: First, the GC can cause unpredictable fluctuations in the performance of a cloud application, which can result in higher rates of SLO violations. Second, GC-related CPU spikes can cause increased performance interference, resulting in slowdowns of co-located tenants. To enable improved testing of GC effects, CloudGC a highly-configurable and GC-oriented benchmarking application for clouds was designed and developed. CloudGC was then used to evaluate the performance of the four GC policies implemented in the IBM JVM. The experimental results suggested that Gencon, the default policy, was the most performant and the least interfering in the majority of the test cases.

Finally, to mitigate performance interference caused by the GC during periods of low loads and idleness, Elastic GC was proposed and tested using CloudGC. Elastic GC detects times of reduced activity and during them, it scales down the runtime’s heap size and the number of CPU cores the GC threads are allowed to use, in conjunction with executing a preemptive GC. The experimental evaluation suggested that the technique of Elastic GC was successful in reducing interference when the application maintained a large object-graph and the runtime used a GC policy that included large stop-the-world and CPU-intensive phases.
9.1 Future Directions

The core contributions of this thesis revolved around performance interference, scaling and SLO satisfaction for clouds and runtimes. Cloud computing is likely to become even more ubiquitous in the future; in conjunction with the ever increasing computational needs of the world, further methods of increasing its efficiency should be researched. Essentially, work needs to be done in packing tenants as close as possible such that their levels of multitenancy and resource sharing are maximized, without violating SLOs. Nevertheless, caution needs to be taken to ensure that privacy and security are not compromised. Consider the Spectre exploit for instance, which was published in January 2018 [71]. Spectre uses speculative execution, which is available in a vast array of modern processors, to gain access to otherwise protected memory. In particular, the attack updates a variable, if a bit in a protected address is on. Even though the CPU cancels the transaction once it finishes performing the branch check and the variable is not actually updated, it is nevertheless, brought into the cache. Thus, subsequently measuring the time this variable takes to be read, reveals the value of the protected memory bit! Various system software patches were published before the writing of this thesis but the extents of this vulnerability remain largely unexplored. For instance, the proposed multitenant application server could be vulnerable to such an attack; further research is required to ensure that a malicious ASaaS tenant cannot use a Spectre-like attack to gain access to sensitive data.
of any other ASaaS tenants—this problem is not intrinsic to ASaaS, Spectre could also be used by a malicious request to access data of other requests even on a simple single-tenant application server. In addition, similar concerns are raised for Type-2 hypervisors, which run multiple VM tenants on a single OS process, and are routinely used in cloud deployments. Further research is needed to ensure security for these types of clouds. Reportedly\textsuperscript{1}, the proposed patches for Spectre can cause significant slowdowns; it remains to be seen how this affects the SLO satisfaction of cloud tenants as well as their performance interference.

Apart from security, new cloud technologies are required to support the rapidly expanding Internet of Things (IoT) domain. IoT devices are outsourcing virtually all of their computations and tend to produce and receive a copious amount of data. To minimize traffic to the centralized cloud nodes, fog computing performs a number of these computations on the edge \cite{35}. With the increasing number of IoT devices an edge needs to support, the experience acquired from virtualization and cloud computing will become a guiding point towards designing, implementing and evaluating the performance of such resource-constrained mini-clouds. In particular, the performance interference lessons gained from this work could be directly applied in fog/edge research.

This thesis also has a number of short-term extensions, some of which are

already being planned. In particular, CloudGC can be used to evaluate further Java runtimes and experimental GC policies in terms of SLO satisfaction. A more ambitious project on this front could be an SLO-aware GC policy, which will need to dynamically change its behavior based on the incoming load, fluctuating SLOs, number of instances and the behavior of any co-located tenants.

Furthermore, mitigating performance interference of runtimes through scaling its parts at specific times is also under investigation. Future work on this front can include scaling the number of mutator threads a runtime requires to best satisfy its SLOs and reducing the resources consumed at startup of cloud applications that are in development mode. The efficacy of machine learning algorithms could also be explored for such optimization tasks: cloud application servers tend to handle load that fluctuates in predictable patterns.

Finally, a driving motor in the fields of Computer Science and Software Engineering is the constantly upgraded and novel hardware and system software. Cloud systems, runtimes and application servers that run on top of these stacks need to be kept up to date in order to best utilize them. Hardware accelerators, such as FPGAs and GPGPUs, are more frequently embedded in clouds; the automatic outsourcing of pieces of work to such devices, needs to be performed securely and with predictable performance interference.
Bibliography


\%2Fcom.ibm.java.lnx.70.doc\%2Fuser\%2Fjava_jvm.html, [Accessed: Feb-2016].


191


[25] Claudio Ardagna, Ernesto Damiani, Fulvio Frati, Davide Rebecchi, and Marco Ughetti, Scalability patterns for platform-as-a-service,


[71] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael


[115] Ryan Sciampacone, Peter Burka, and Aleksandar Micic, *Garbage collection in WebSphere application server V8, part 2: Balanced garbage*


[133] Tom Watson, *How dense is the cloud of OSGi?*, [Online].
https://www.eclipsecon.org/na2014/sites/default/files/slides/EclipseCon\%202014\%20-%\%20Multitenant\%20OSGi.pdf,
[Accessed: Feb-2016].


Vita

Panagiotis Patros

Master in Computer Science, University of New Brunswick, 2014
Degree in Informatics and Telecommunications, National and Kapodistrian University of Athens, 2010

Patent Disclosures


P. Patros, K.B. Kent and M. Dawson, Dynamic SLA Aware Automatic Microscaling for Application Servers on Multitenant Systems, CA8 2016 0552, Dec 2016, To be defensively published by IBM

P. Patros, M. Dawson and K.B. Kent, Minimizing Interference and Improving Locality for Container-Multitenancy through Limiting Resources on Startup, CA8 2016-0403, Nov 2016, Defensive publication by IBM, IPCOM 000252861 (Feb 2018)

P. Patros, M. Dawson and K.B. Kent, SLA-Aware Thread-Pool and Request Handling for Application Servers, CA8-2016-0250, July 2017, Defensive publication by IBM, IPCOM 000252550 (Jan 2018)

P. Patros, E. Aubanel, D. Bremner and M. Dawson, Efficient data collection for Java Util Concurrent locks and structures, Sep 2013, Defensive publication by IBM, IPCOM 000239095 (Oct 2014)

Publications


S. Seeley, V. Sankaranaryanan, Z. Deveau, P. Patros and K.B. Kent, Simulation-Based Circuit-Activity Estimation for FPGAs Containing Hard Blocks, Proceedings of the 28th International Symposium on Rapid System Prototyping (RSP@ESWeek 2017), Seoul, South Korea, pp. 36-42, Oct 19-20, 2017


P. Patros and K.B. Kent, Automatic Detection and Elision of Reset Sub-Circuits, Proceedings of the 27th International Symposium on Rapid System Prototyping (RSP@ESWeek 2016), Pittsburgh, USA, pp. 1-7, Oct 6-7, 2016


**Theses Publications**


Academic Instructor

Introduction to Programming (in Java), UNB CS1073, Winter 2018
Introduction to Programming II (in Java), UNB CS1083, Fall 2017
Computability and Formal Languages, UNB CS2333, Winter 2016
Introduction to Programming (in Java), UNB CS1073, Fall 2015
Introduction to Computer Graphics, UNB CS4735, Fall 2014

Experience

Contract Computer Specialist, Fenix Media, 2015
Contract Computer Specialist/Consultant, STiX, 2010-2012
Greek Army, 2009-2010, Signals’ Corporal in Reserve
Computer Specialist/Consultant, OETA, 2007-2009