



## ACTiCLOUD: ACTivating resource efficiency and large databases in the CLOUD

Project No: 732366

H2020-ICT-2016-1

### D2.4: Distributed Cloud Resource Manager v2.0

Due date of deliverable:	M32 (2019/08/31)
Actual submission date:	M34 (2019/10/25)

**Executive summary:** Deliverable D2.4 provides the final version of ACTiCLOUD's cloud resource manager, named as ACTiManager, that consists of functional components, placement algorithms, models for workload classification, interference detection, and performance prediction. While generic in concept and design, ACTiManager is implemented on top of and fully integrated with OpenStack cloud management platform. The deliverable consists of the software source code and its documentation.

**List of authors:**

Authors	Affiliation
Georgios Goumas, Vasileios Karakostas, Konstantinos Nikas, Dimitrios Siakavaras, Stratos Psomadakis, Stefanos Gerangelos, Ioannis Papadakis, Nikela Papadopoulou	ICCS
Ewnetu Bayuh Lakew, Petter Svärd, Simon Kollberg	UMU

<b>Dissemination Level</b>	<b>X</b>	<b>PU (Public)</b>
		PP (Restricted to other programme participants)
		RE (Restricted to a group specified by the consortium)
		CO (Confidential, only for members of the consortium)
	Where restricted, access granted to:	
<b>Nature</b>		R (Report)
		P (Prototype)
		D (Demonstrator)
	<b>X</b>	<b>O (Other)</b>

<b>Review Status</b>		Draft
		WP Leader accepted
		QA approved
	<b>X</b>	<b>Coordinator accepted</b>

**Revision History:**

Version	Author(s) (Affiliation)	Notes
0.1	Vasileios Karakostas (ICCS)	Initial ToC
0.2	Georgios Goumas (ICCS)	Initial content in Sections 1 and 2
0.3	All authors	Content updated
0.4	Ying Zhang (MDBS), Michail Flouris, Stelios Louloudakis (ONAPP)	Document reviewed
0.5	All authors	Address comments and suggestions
1.0	Georgios Goumas, Vasileios Karakostas (ICCS)	Final version to be submitted to EC

**ACTiCLOUD Consortium:**

Participant No	Participant organisation name	Short name	Country
1 (Coordinator)	Institute of Communication and Computer Systems	ICCS	Greece
2	Numascale AS	NSCALE	Norway
3	Kaleao Limited	KALEAO	UK
4	OnApp Limited	ONAPP	Gibraltar
5	University of Manchester	UNIMAN	UK
6	MonetDB Solutions B.V.	MDBS	Netherlands
7	Neo Technology	NEO	Sweden
8	UMEA University	UMU	Sweden



NUMASCALE



**Confidentiality:**

This document contains proprietary and confidential material of certain ACTiCLOUD contractors, and may not be reproduced, copied, or disclosed without appropriate permission. The commercial use of any information contained in this document may require a license from the proprietor of that information.

THIS DOCUMENT IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES

(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

**Table of Contents**

<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	Purpose of this document.....	9
1.2	Document structure .....	9
<b>2</b>	<b>Relevance and Motivation</b>	<b>10</b>
2.1	Relevance to ACTiCLOUD objectives, business scenarios, and use cases .....	10
2.2	Background .....	10
2.3	Motivation .....	11
<b>3</b>	<b>ACTiManager design and architecture</b>	<b>13</b>
3.1	Design Principles .....	13
3.2	Overview .....	15
3.3	Detailed design and architecture.....	17
3.3.1	ACTiManager.internal.....	18
3.3.2	ACTiManager.external .....	28
3.3.3	ACTiManager.multicloud (Cross-Site Offloading) .....	31
3.3.4	ACTiManager - Cloud Manager roles and interaction .....	31
3.4	ACTiManager Operation .....	32
3.5	Events .....	33
3.6	Actions .....	34
3.7	Models .....	34
3.8	The Lifecycle of a VM under ACTiManager .....	35
<b>4</b>	<b>ACTiManager invocation scenarios</b>	<b>37</b>
4.1	The VM creation scenario.....	37
4.2	The interference detection scenario .....	37
4.3	The overload, underload, and fragmentation detection scenarios .....	38
4.4	Imbalance detection scenario .....	39
4.5	Application under performance .....	39
4.6	Cross-site offloading scenario .....	39
<b>5</b>	<b>Module description</b>	<b>40</b>
5.1	Overview .....	40
5.2	Internal and External sub-modules.....	40
5.2.1	Internal .....	40
5.2.2	External.....	41

5.2.3	Communication mechanism between Internal and External sub-modules .....	41
5.2.4	Communication support between ACTiManager.internal and hypervisor .....	42
5.3	Information Aggregator Component .....	42
5.3.1	Internal .....	42
5.3.2	External.....	43
5.3.3	Performance Agent .....	43
5.4	Modeler Component.....	43
5.4.1	Internal .....	43
5.4.2	External.....	46
5.5	Decision maker.....	47
5.5.1	Internal .....	47
5.5.2	External.....	47
5.6	Interaction with OpenStack .....	48
<b>6</b>	<b>Documentation / User manual</b>	<b>49</b>
6.1	ACTiManager.internal.....	49
6.2	ACTiManager.external .....	49
6.3	ACTiManager.multicloud (Cross-site Offloading).....	49
6.4	Characterization agent .....	50
6.5	Communication mechanism.....	50
6.6	Performance Agent.....	50
6.7	Libvirt.....	51
6.8	OpenStack modifications for Placement.....	51
<b>7</b>	<b>Other Components</b>	<b>52</b>
7.1	Cache Partitioning .....	52
7.2	Managing and Redistributing Huge pages.....	53
7.3	Monitoring Active Working Set .....	55
<b>8</b>	<b>Summary</b>	<b>56</b>
<b>9</b>	<b>References</b>	<b>57</b>

**Figures**

Figure 3.1: ACTiManager” follows a hierarchical approach, distinguishing the management of resources at node level (ACTiManager.internal) and at site level (ACTiManager.external). .....	16
Figure 3.2: General architecture and operation of the “ACTiManager.internal” component that manages resources within a single node.....	19
Figure 3.3: Pricing models for gold and silver applications.....	24
Figure 3.4: Server model.....	25
Figure 3.5: Core mapping, Huge VM, as done by the default scheduler used in KVM. ....	27
Figure 3.6: General architecture and operation of the “ACTiManager.external” component that manages resources of a cloud site and across sites.....	29
Figure 3.7: Coarse assumption made by ACTiManager.external for vCPU placement within a server. g is used for gold applications and s for silver applications. ....	29
Figure 3.8: Interaction among multiple ACTiCLOUD sites. ....	31
Figure 3.9: Interaction between ACTiManager and OpenStack. ....	32
Figure 3.10: The lifecycle of a VM under ACTiManager. The blue boxes represent the different states that a VM can be in, while the white boxes denote the models that are used when transitioning from the current state (denoted by the dotted line) to the next state.....	36

**List of Abbreviations and Acronyms**

<b>Abbreviation / Acronym</b>	<b>Meaning</b>
AMQP	Advanced Message Queuing Protocol
CAPEX	Capital Expenditure
CPU	Central Processing Unit
CSPs	Cloud Service Providers
DP	Decision Period
ESD	Estimated Slowdown
IaaS	Infrastructure-as-a-Service
IOPS	Input/Output Operations per second
IPC	Instructions per Cycle
KVM	Kernel-based Virtual Machine
LLC	Last level Cache
MPKI	Misses per Kilo Instructions
MQTT	Message Queuing Telemetry Transport
NUMA	Non Uniform Memory Access
OPEX	Operational Expenditure
PCA	Principal Component Analysis
pCPU	Physical CPU
QoS	Quality of Service
SDN	Software Defined Network
SLA	Service Level Agreement
SLI	Service Level Indicator
SLO	Service Level Objective
STOMP	Streaming Text Oriented Messaging Protocol
SO	Strategic Objective
THP	Transparent Huge Pages
TLB	Translation Lookaside Buffers
vCPU	Virtual CPU
VM	Virtual Machine



# 1 Introduction

ACTiCLOUD’s vision is to develop a novel cloud architecture that breaks the existing scale-up and share-nothing barriers, and enable the holistic management of physical resources both at the local cloud site and at the distributed levels. ACTiCLOUD targets drastically improved utilization and scalability of resources. This will ultimately translate to:

1. significant cost and performance improvements for Cloud Service Providers (CSPs),
2. higher performance, stability, and lower pricing for cloud applications,
3. enhanced flexibility and scalability of cloud resources for intensive database applications that have until now faced tough challenges in covering their resource demands from existing cloud offerings.

ACTiCLOUD aims to enhance the viability of cloud deployment scenarios through enhancement of the various technology ingredients, i.e., the hypervisor, the cloud manager, system libraries, language runtimes, and database systems, with a novel and holistic set of mechanisms and policies built on top of these new-generation computing system architectures. Therefore, ACTiCLOUD will enable the creation of distributed, hyper-converged, “share-everything”, resource scale-out cloud platforms to broaden the applicability of cloud technologies across more markets through richer and more cost effective application deployments.

## 1.1 Purpose of this document

This document accompanies ACTiCLOUD Deliverable D2.4: “Distributed Cloud Resource Manager v2.0”, the software module implementing the functionality of ACTiManager [ACTi\_D1.2]. ACTiManager is an advanced resource manager that supports the core goals of ACTiCLOUD towards resource-efficient IaaS cloud offerings. The core functionality of ACTiManager along with the basic components was described in D2.2: “Distributed Cloud Resource Manager v1.0” [ACTi\_D2.2]. This version of deliverable (v2.0) implements the final version of ACTiManager and is delivered as a software module. This document presents the design and the updated components for resource management and workload characterization. To make this document self-containing we include parts of the description found in D2.2, but updated to the final version.

## 1.2 Document structure

The document is organized as follows: Section 2 describes the relevance of ACTiManager to the ACTiCLOUD objectives, background information, and the motivation for including ACTiManager within the stack of ACTiCLOUD. Section 3 describes the design principles of ACTiManager and presents its design and architecture. Section 4 discusses in more detail the various invocation scenarios of ACTiManager. Section 5 describes in detail the software components of ACTiManager. Section 6 provides documentation information about the software module. Section 7 describes other components that have been developed for generic platforms which have not been currently integrated with other components of the ACTiCLOUD stack, but are planned to be integrated in the near future. Finally, Section 8 concludes this document.

## 2 Relevance and Motivation

This section describes the relevance of ACTiManager to the ACTiCLOUD objectives, business scenarios, and use cases. Then, it provides background information regarding the server architectures. Finally, it describes the motivation for including ACTiManager within the stack of ACTiCLOUD.

### 2.1 Relevance to ACTiCLOUD objectives, business scenarios, and use cases

ACTiManager is one of the most critical components in the ACTiCLOUD architecture as it plays a central role in the realization of ACTiCLOUD's objectives towards next generation IaaS platforms [ACTi\_D1.1]. ACTiManager enhances state-of-the-art cloud management approaches with capabilities to:

- perform more effective resource allocation, increasing system throughput (Strategic Objective S01.1 on resource efficiency),
- equip applications with more strict performance guarantees (Strategic Objective S01.2 on performance stability).

In addition, as part of the ACTiCLOUD architecture, ACTiManager supports the delivery of scale-up and scale-out resources offered by the underlying hardware and hypervisor layers (Strategic Objective S02.1 on scalability in resource provisioning) and interacts with applications to respond to specific-tailored, dynamic resource requests (Strategic Objective S02.2 on elasticity in resource provisioning).

Moreover, ACTiManager is designed to directly support ACTiCLOUD's business scenarios [ACTi\_D1.1, ACTi\_D1.2] as summarized below:

- Business scenario 1: Effective consolidation for increased revenue and reduced TCO,
- Business scenario 2: Workload prioritization,
- Business scenario 3: Hosting larger workloads,
- Business scenario 4: Collaboration with sibling cloud sites,
- Business scenario 5: Enhanced dependability and availability.

Finally, the direct focus of ACTiManager is to support "Use case 1: Execution of typical cloud applications". In addition, ACTiManager is expected to play an important role in enabling the Use Cases 2-7 that are more relevant to data-driven applications [ACTi\_D1.1, ACTi\_D1.2].

### 2.2 Background

Depending on the type of hardware, huge performance variability for applications may be observed due to interference in shared resources and how the virtual resources are aggregated and mapped onto the physical resources. The resource manager should avoid and mitigate contention for shared resources. In addition, the resource manager should consider and optimize locality and proximity of the resources that constitute the virtual machine (VM) where the application runs.

**Performance issue due to Interference**

Typical cloud servers are organized in a number of NUMA sockets (or clusters of CPUs<sup>1</sup>) each hosting CPUs with multiple cores and sharing parts of the memory hierarchy. Figure 2.1 shows an example of a server with two sockets, four cores per socket, with private L1 caches, shared L2 caches between cores and a per socket L3 cache (Last Level Cache – LLC). As applications running on this machine may share multiple levels of the memory hierarchy (caches or memory links), interference may occur.

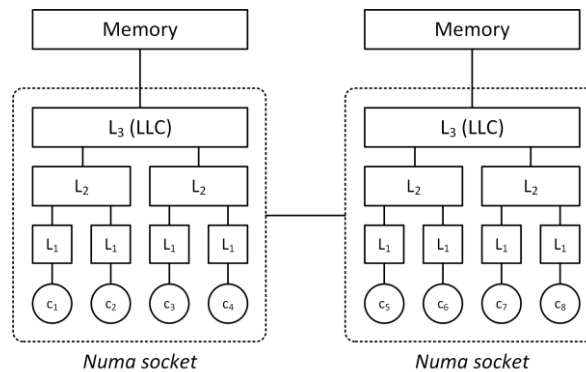


Figure 2.1: Generic architecture of a cloud server.

**Performance issue due to locality**

Performance optimization for locality can be achieved by making each thread access local off-chip memory upon cache misses [SLN12]. Performance limitation due to remote memory accesses may limit application scalability [TSR08], especially as multiple servers become one in the case of Numascale, the effects of collocating many applications and emergent behaviour become more prominent. As a result, application performance may vary depending on how the virtual resources are mapped to physical resources.

**2.3 Motivation**

The cloud paradigm has created a rapid technology shift from small, in-house private infrastructures to large-scale, public or private datacenters. To accommodate the unprecedented demand for cloud computing and maximize clients' satisfaction, cloud service providers (CSPs) have been relying on continuous infrastructure expansions and on simplistic policies to manage resources. While this was a reasonable strategy for early adoption, it fails to sustain the growing demand for cloud computing.

Recent studies have shown that resource utilization ranges from 10% to 50% [DK14, LCG15] in most cloud datacenters. The root cause for resource waste is the conservative allocation policies of CSPs. The co-existence of multiple workloads within the same physical server can create performance degradation and unpredictability due to contention in shared resources (e.g., CPUs, caches, memory links, etc.) [ZBF10, PNK17, TMS11, HGA14, MSB10, DK13, KJL18]. A straightforward and frequently preferred solution attempt is to completely isolate latency-critical virtual machines

<sup>1</sup> As is the case for Exynos that equip KMAX compute resources.

(VMs) either by granting them an entire physical server or large parts of it (e.g., a NUMA node [RKG13]). Clearly, such attempts leave precious resources underutilized in the fear of interference.

The main approaches towards dealing with interference are avoidance, i.e., allocate resources avoiding interference, and mitigation, i.e., apply isolating actions when interference is detected. Several research resource managers have incorporated interference avoidance and/or mitigation in their operations [DFB12, DK13, YBM13, NVN13, LCG15, RD18, NKG10]. However, those systems are based on constraints that limit their applicability to operational cloud environments. In particular, some systems rely on offline profiling information that needs to be collected before VM execution [NKG10, RKG13, DK13, YBM13, RD19], and/or requires the availability of a Quality-of-Service (QoS) reporting mechanism from the applications [YBM13, LCG15]. In addition, some systems lack support for multi-phase applications [RD18], i.e., applications that exhibit different behavior during their execution lifetime, or need to apply online probes to support accurate predictions [YBM13, NVN13]. Finally, prior systems either seek to guarantee the performance of latency-critical VMs or to increase the total system utilization, without considering how application performance translates to profit from the CSPs' perspective.

ACTiManager is an end-to-end interference-aware cloud resource manager that follows a hierarchical two-tier approach to optimize the allocation and utilization of resources in both node and site levels.

Furthermore, the “share-nothing” PC system architecture used in datacenters today confines resource allocation to the physical bounds of servers. As a result the current cloud setup only hosts applications that are either resource conservative or capable of requesting resources in the “scale-out” fashion, i.e., they can be easily decomposed into large numbers of loosely connected entities adhering to the “share-nothing” paradigm [LPM17]. However, large classes of important applications are resource hungry and do not scale out gracefully for simplicity, correctness, programmability, performance and cost effectiveness reasons. These applications are characterized as “scale-up”, since they need to execute on a single server under the “share-anything” paradigm.

To make matters worse, typical cloud configurations involve sites that are geographically distributed and disjointly managed either due to operational needs or scalability restrictions. Solutions such as WL2 [CLL15] have been proposed recently that leverage Software Defined Networking (SDN) to provide a scalable, high-performance, multi-datacenter layer-2 network architecture, thus enabling tighter cross-site interactions.

ACTiManager builds on top of existing technological development to provide a set of mechanisms and policies that enables offloading of application between cloud sites that are geographically distributed to alleviate temporary site overload. In this way, ACTiManager enabled platforms are able to support policies enforcing geo-based load balancing by temporarily offloading selected applications from highly overloaded sites to less overloaded sites.

In summary, the goal is to provide a cloud manager that:

- performs both interference avoidance and interference mitigation,
- allocates resources considering the physical machine's hardware configuration,
- undertakes priority-based VM placement, and
- performs cross-site offloading to mitigate site overload.

### 3 ACTiManager design and architecture

In this section we describe the design principles of ACTiManager and present its design and architecture.

#### 3.1 Design Principles

ACTiManager is designed with the following principles in mind<sup>2</sup> [KGL18]:

1. To operate in the typical closed-loop control fashion based on: (a) monitoring and information aggregation, (b) information processing and modeling, and (c) decision making and actuation, leading respectively to the three core components of the module:
  - the Information Aggregator,
  - the Modeler, and
  - the Decision Maker.
2. To be scalable in large-scale cloud installations, operating at various levels, including node level, cloud site level, and inter-site. For this reason, ACTiManager follows a modular hierarchical design (see Figure 3.1), split into two sub-modules:
  - ACTiManager.internal whose goal is to manage resources at a fine-grained level within a node (e.g., mapping of virtual CPUs on physical CPUs, allocation of memory, etc.), and
  - ACTiManager.external whose goal is to manage resources within and across cloud sites, enforcing high-level policies for placement, load balance, increased utilization, efficient consolidation, energy efficiency, cross-site offloading, etc.
3. To minimize the modifications to an existing cloud management system and to operate as an “out-of-the-box” add-on component that can be plugged-in and out of an existing installation at will. Towards this direction, we put significant effort in interfacing with existing, well established components of the core cloud manager (OpenStack<sup>3</sup> in our case), and minimizing the necessary changes to the core cloud manager itself for integration with ACTiManager.
4. To support the relevant ACTiCLOUD’s strategic objectives, business scenarios, and use cases:
  - ACTiManager relies primarily on online, but also supports offline, characterization and classification of the VMs (or applications) and incorporates in their feature list characteristics that describe a VM’s potential to create interference (i.e., “noisy/quiet” VM) or suffer from interference (i.e., “sensitive/insensitive” VM)

---

<sup>2</sup> Vasileios Karakostas, Georgios Goumas, Ewnetu Bayuh Lakew, Erik Elmroth, Stefanos Gerangelos, Simon Kolberg, Konstantinos Nikas, Stratos Psomadakis, Dimitrios Siakavaras, Petter Svard, and Nectarios Koziris. “Efficient Resource Management for Data Centers: The ACTiCLOUD approach”. In 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS 2018).

<sup>3</sup> <https://www.openstack.org/>

regarding the use of resources, by the co-location of another VM within the same executing node:

- A "noisy" VM uses a significant part of some shared resources that might affect the performance of another VM that is co-located with. An example of "noisy" application for the shared last level cache (LLC) is a streaming application whose performance is not improved by having more cache; but, because it uses as much cache as it has access to, it might affect the performance of other applications that share that cache with. In contrast, a "quiet" VM uses shared resources in a modest way. An example of "quiet" application for the shared LLC is an application that actively uses only a small portion of the shared LLC, leaving the rest of LLC underutilized. Hence, the co-location of a quiet VM with another VM will not cause performance interference to that VM.
- A "sensitive" VM is a VM whose performance depends on the resources that are assigned to it. An example of "sensitive" application regarding the shared LLC is a cache-friendly application, whose performance improves as it gets more cache. An "insensitive" VM is a VM whose performance is independent of the resources that are assigned to it. An example of "insensitive" application regarding the shared LLC is a streaming application, whose performance is not improved by having more cache. Co-locating "noisy" applications with "sensitive" applications can severely affect the performance of the sensitive applications due to interference in shared resources.
- ACTiManager exploits online anomaly detection in the VMs' performance to mitigate performance issues. To support anomaly detection, ACTiManager incorporates in the feature list of each VM a "healthy state" model that describes the VM's behavior when executing in isolation without any interference. Note that ACTiManager operates in an application-transparent way for both classification and anomaly detection, i.e., ACTiManager relies on low-level system metrics from hardware monitoring facilities and events within the hypervisor.
- To dynamically classify the VMs for interference avoidance and to extract the "healthy state" model for interference detection and mitigation, ACTiManager logically splits the site infrastructure, similarly to [NVN13], into:
  - (i) a small number of nodes operating as "laboratory" nodes, and
  - (ii) the rest (vast majority) of the site nodes as "production" nodes.

More specifically, ACTiManager uses the "laboratory" node(s) for quickly classifying/characterizing the VMs and extracting the aforementioned information (i.e., "noisy/quiet" or "sensitive/insensitive" labels) in an execution environment that is free from any source of interference. In addition, ACTiManager uses the "laboratory" node(s) for extracting the "healthy state" model that is used for interference detection. Based on the classification,

ACTiManager then selects the "production" node where each VM will continue its execution. Note that in the "laboratory" node more than one applications can be classified. The laboratory environment may consist of a single isolated NUMA node. In this way, multiple applications can be classified concurrently. The rest of the compute nodes, i.e., the vast majority of the resources, are treated as usual compute or "production" nodes. More details about the lifecycle of a VM are provided in Section 3.8.

- Cloud infrastructures host a wide variety of applications from various domains and with diverse configurations, execution characteristics, and QoS demands. To capture this diversity, cloud service providers and prior research distinguish two major classes of applications, i.e., latency critical and best effort [LCG15, NPG19, YBM13].

In similar spirit, ACTiManager also considers that the cloud site administration may distinguish between two types of services, i.e., gold and silver:

- (i) "gold" instances, i.e., high-priority, latency-critical VMs that expect to enjoy guaranteed performance and thus they are priced higher, and
- (ii) "silver" instances, i.e., low-priority, batch VMs that expect to enjoy best-effort performance, potentially with a different billing policy, priced lower than gold VMs.

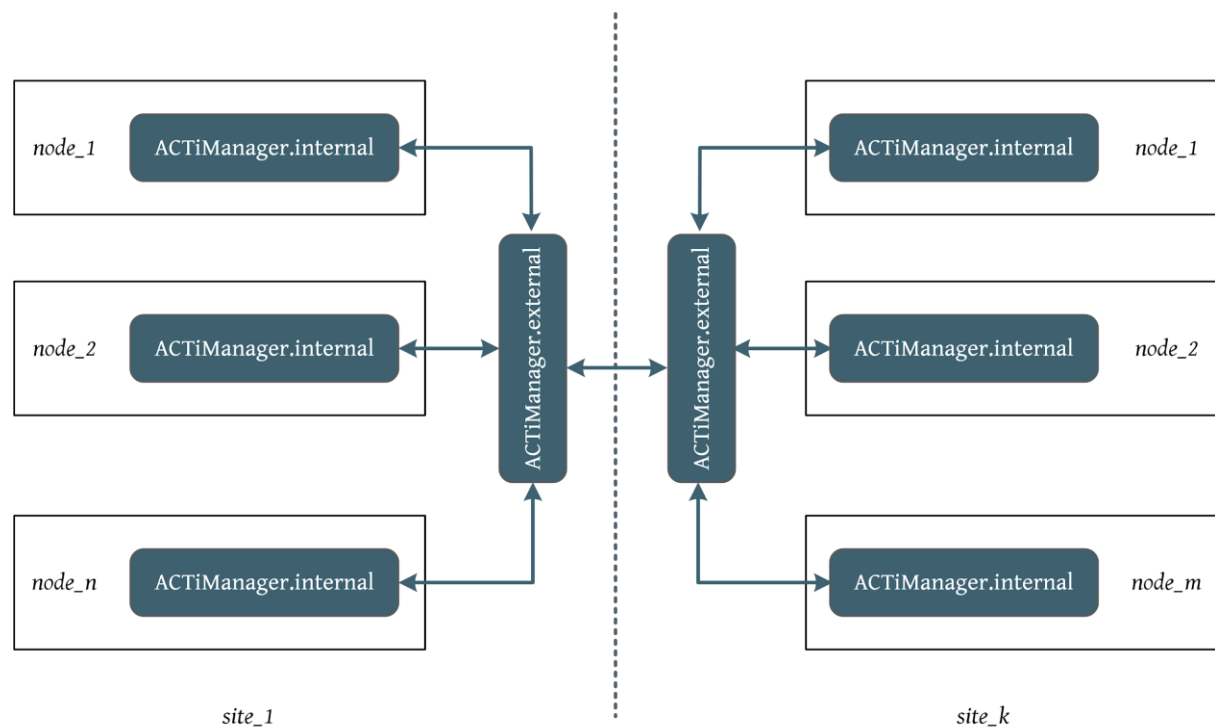
The reader may refer to [ACTi\_D1.2 - page 15] for a more detailed analysis of Business Scenario 2 regarding prioritization. ACTiManager takes this prioritization into account when placing VMs and assigning resources, as explained next in Section 3.3.

- ACTiManager assumes that some applications may incorporate special functionality to expose their desired metric of interest and achieved Quality of Service (QoS) via the Performance Agent component (see Sections 5 and 6). Note that this is not a requirement for ACTiManager to operate. However, if this functionality is provided by an application, ACTiManager is able to utilize it to act more efficiently (i.e., faster, more accurately) towards maintaining the desired QoS of applications.
- ACTiManager assumes that applications that can be potentially offloaded to sister site(s) are stateless, as stateful-migration across different sites is too expensive. This assumption does not hold for migrating VMs within the same site, i.e., ACTiManager may select any VM, based on its algorithms and policies, for migration within the datacenter or the node.

## 3.2 Overview

As shown in Figure 3.1 and described above, a specific ACTiManager instance takes care of resource orchestration internally within each node (ACTiManager.internal), and a specific ACTiManager component orchestrates resources across nodes within a cloud site and between distributed cloud sites (ACTiManager.external). Depending on the underlying architecture and the setup of the cloud

site, a node in one setup may be substantially different compared to a node in another setup. To cope with this issue, we refer to a node as the atomic compute unit that is managed by a single hypervisor instance. Thus, in the frame of ACTiCLOUD project a node in the Numascale system is a rack of servers interconnected with Numaconnect (Figure 4.2 in [ACTi\_D1.2]), while a node in the KMAX platform is an Exynos server (Figure 4.3 in [ACTi\_D1.2]), both managed by one instance of the hypervisor.



**Figure 3.1:** ACTiManager” follows a hierarchical approach, distinguishing the management of resources at node level (ACTiManager.internal) and at site level (ACTiManager.external).

Regardless of its level of operation (node or site), ACTiManager consists of the following three components:

- The Information Aggregator component that is responsible for collecting information from various monitoring facilities.
- The Modeler component that is responsible for providing models: (i) to characterize and classify the execution of the applications as “noisy/quiet” or “sensitive/insensitive” regarding the use of resources, (ii) to extract the “healthy state” model and perform anomaly detection due to interference, and identify imbalance, overload, etc., and (iii) to predict the impact of various actions (e.g., consolidation effect on the already executing application, probability to meet performance guarantees, migration feasibility, and migration time).
- The Decision Maker component that decides about the optimized resource allocation and initiates the relevant actions, including placement, prioritization, consolidation,



offloading, and interference mitigation of the running VMs.

ACTiManager can be integrated within a typical cloud software stack that includes a cloud manager and a virtualization support layer.

For the development of ACTiManager we use Openstack (Pike version<sup>4</sup>), as the de-facto, open source IaaS cloud manager and controller for Virtual Machines (VMs) and make use of its mechanisms to create and manage VMs (e.g., through migration). The execution of virtual machines is enabled by the hypervisor (e.g., OnApp's MicroVisor, KVM<sup>5</sup>, Xen<sup>6</sup>) that provides a multitude of features to manage VMs at server level, e.g., by pinning VMs on physical resources, moving VMs within a server, migrating VMs, etc.

Note that the operation of ACTiManager is orthogonal to the cloud software platform. ACTiManager is currently integrated with the aforementioned stack, but it can also be integrated with other technologies regarding resource management, e.g., Kubernetes<sup>7</sup>.

This deliverable describes ACTiManager on top of OpenStack and KVM assuming generic commodity NUMA (non-uniform memory access) x86 servers. Section 5 describes particular components for the Numascale and KMAX hardware platforms provided by the project's partners NSCALE and KALEAO respectively. The integration of ACTiManager with OnApp's MicroVisor for the final ACTiCLOUD prototype will be documented in the deliverable D4.4 "ACTiCLOUD Final Prototype".

Next we describe the role of each subcomponent in more detail and its relevance to the level of operation, and provide more detail on the interaction between the ACTiManager and the OpenStack core cloud manager.

### 3.3 Detailed design and architecture

ACTiManager is an end-to-end interference-aware cloud resource manager that targets both interference avoidance and interference mitigation<sup>8</sup> [PGS19]. With ACTiManager, VMs start their lifetime in a cloud environment within a protected, isolated island, which typically is a NUMA socket (for VMs that require less resources than a NUMA socket) or an entire server. This unleashes the potential to collect online information in an interference-free environment. We characterize the behavior of the VM using machine-learning techniques and create the resource fingerprint of the VM regarding its potential to create interference, i.e., noisy or quiet, or suffer from interference, i.e., sensitive or insensitive. Similarly, we build a model to describe its healthy state which later on will be utilized to detect interference. From that point on, the VM can be placed in a symbiotic environment and proceed with its execution under ACTiManager's supervision. Note that a default interference-aware policy would place VMs (or gold VMs) in an isolated environment

---

<sup>4</sup> <https://www.openstack.org/software/pike/>

<sup>5</sup> [https://www.linux-kvm.org/page/Main\\_Page](https://www.linux-kvm.org/page/Main_Page)

<sup>6</sup> <https://xenproject.org/>

<sup>7</sup> <https://kubernetes.io/>

<sup>8</sup> Stratos Psomadakis, Stefanos Gerangelos, Dimitrios Siakavaras, Ioannis Papadakis, Marina Vemmou, Aspa Skalidi, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Georgios Goumas. "ACTiManager: An end-to-end interference-aware cloud resource manager". In 20th International Middleware Conference Demos and Posters (Middleware 2019).

during their entire execution. Instead, ACTiManager uses the isolated environment only in the beginning of the VM's execution. Hence, we believe that it comes at almost no cost to first characterize VMs and then place them appropriately in the datacenter. Alternatively, the user may provide information regarding classification or the ACTiManager may store such information from previous runs of a user's instances to relax the need for characterizing all VMs.

Our goal with ACTiManager is a practical interference-aware resource manager that:

- performs both interference avoidance and interference mitigation,
- requires no offline application profile,
- operates in an application transparent way,
- avoids artificial interference through probes, and
- optimizes for datacenter profit.

### 3.3.1 ACTiManager.internal

ACTiManager.internal is responsible for managing resources within a single node. To this end, it collects information about the resource utilization of the running VMs within the node, performs sanity checks (e.g., detects interference, server overload, underperformance of VMs/ applications) and takes relevant actions locally (e.g., remaps virtual (vCPUs) to physical (pCPUs) cores, moves VMs' data from one NUMA socket to another within the node). In case the local actions are unsuccessful or insufficient to enforce the desired policy, it informs the ACTiManager.external sub-module.

ACTiManager.internal comes in two flavors:

- The laboratory version analyzes newly arrived VMs in an environment free of interference, monitors them, characterizes them in terms of their potential to create or suffer from interference, and creates a model that describes the VM's healthy state.
- The standard ACTiManager.internal version places (pins) the VMs in specific cores and NUMA memory nodes, monitors their performance and takes actions (like repinning, migrating memory or informing ACTiManager.external) when performance anomalies in execution are observed.

The general architecture and operation of the ACTiManager.internal sub-module is shown in Figure 3.2.

The Information Aggregator collects information from three major sources:

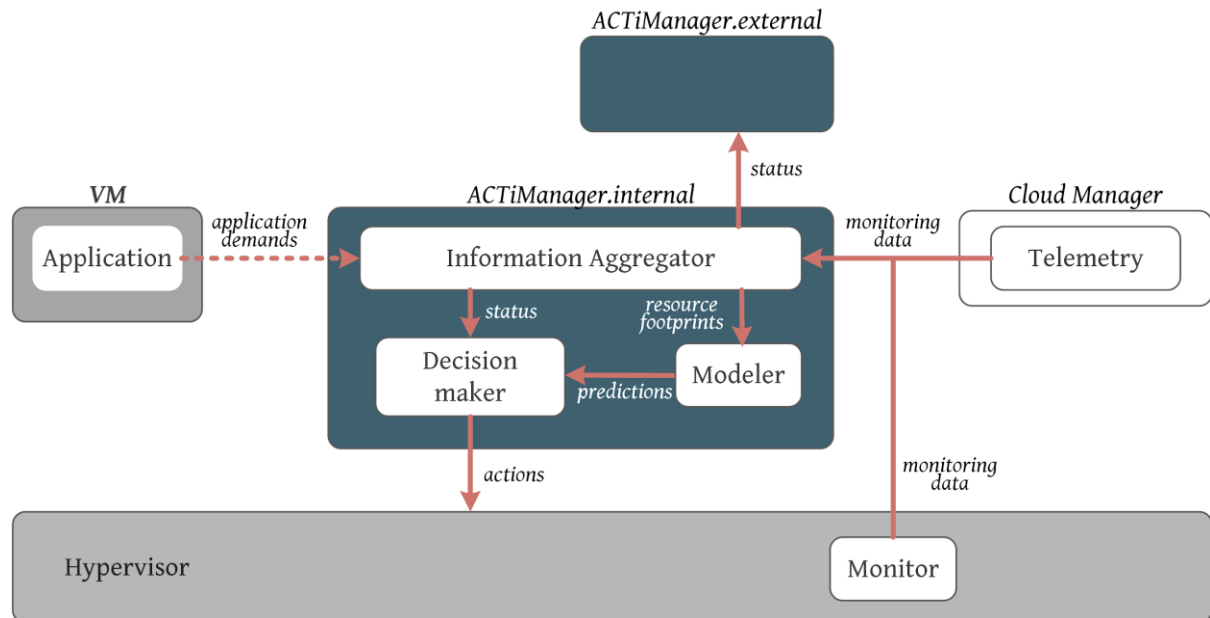
- The Telemetry<sup>9</sup> source that collects all the monitoring information that the standard agent of the Cloud Manager provides. In the context of OpenStack, the Telemetry subcomponent can refer to Ceilometer<sup>10</sup>.
- The Monitor facilities within the hardware or the hypervisor that collect and make available information from the hardware monitoring facilities and events within the hypervisor, e.g., instructions per cycle (IPC), cache misses per kilo instructions (MPKI), CPU utilization, input/output operations per second (IOPS), etc.

---

<sup>9</sup> <https://wiki.openstack.org/wiki/Telemetry>

<sup>10</sup> <https://docs.openstack.org/ceilometer/latest/>

- The application demands reported with special interfaces directly to the ACTiManager. This is an optional source of information (denoted by the dotted line), as ACTiCLOUD does not demand this feature from applications, but can utilize it if it is available; for example MonetDB provides such information to ACTiManager.



**Figure 3.2:** General architecture and operation of the “ACTiManager.internal” component that manages resources within a single node.

The Information Aggregator sends status information to the upper level of the ACTiManager hierarchy and also provides this information to the Modeler.

The Modeler serves two critical roles:

- It performs workload classification based on the information gathered from the Information Aggregator.
- It predicts the potential for improving the profit (cost-benefit) from the CSP perspective, by taking into consideration the characteristics of the VMs, the potential for performance improvement, the cost of potential interference, and the cost of performing an action.

The Decision Maker is the heart of the optimization processes that take place in ACTiCLOUD. Both the Modeler and the Information Aggregator support the Decision Maker component. The Decision Maker tries to optimize resource allocation at the node level for profit or performance maximization by enforcing the policies and priorities that are defined within ACTiCLOUD, and by deciding on the appropriate configurations of the running VMs within the node. To enforce its decisions, the Decision Maker in the ACTiManager.internal requests actions from the Hypervisor. Such actions include (re-)pinning the vCPUs and memory of the VMs within the node.

### 3.3.1.1 Laboratory version of ACTiManager.internal

ACTiManager targets both interference avoidance and mitigation, but operates with no prior information about the execution behavior of an application, i.e., the application starts its normal

execution within the cloud datacenter and carries no profiling information. However, to apply these two policies we need to feed reliable information regarding the application's resource footprint into fairly accurate prediction models to support effective decision making.

Specifically, of particular interest in our case is the potential of an application to create or suffer from interference and its resource fingerprint on critical resources when executed in isolation.

To collect this information, VMs start their lifetime in the protected environment of the "laboratory" node where no interference occurs. In the current implementation of ACTiManager, a VM is placed within an isolated NUMA island, but other options are also feasible depending on the VM's characteristics, such as the VM could be placed in multiple isolated NUMA islands or in an entire isolated server.

Having VMs start their normal operation in a quiet neighborhood unleashes a big opportunity to implement an elegant and practical resource manager that both avoids and mitigates interference. Note that this initial state would be the steady state of a straightforward incorporation of interference-awareness to a state-of-practice resource manager like Openstack, that would leave gold VMs alone in a NUMA socket for their entire lifetime and schedule silver applications wherever in the datacenter, as long as they stay away from gold ones. While in the protected environment, ACTiManager collects performance monitoring events that are utilized to characterize the application in terms of interference and record its "healthy state" as discussed next.

#### Interference model

To characterize an application regarding its potential to participate in interference incidents (i.e., to suffer from or create interference), we seek a simple, fast and resource-saving approach. Prior research work has identified a large number of potentially relevant application classes. For example, Xie and Loh proposed an animal classification approach [XL08] and Lin et al. worked on a classification scheme that uses four colors [LLD08]. However, such techniques require information that needs to be collected from static analysis (e.g., memory reuse patterns of applications known as stack distance profiles, or comparison of execution with different cache sizes) or from additional hardware. On the other extreme, one can classify memory-intensive applications using the LLC miss rate [BZF10, KGB13]. However, LLC miss rate on its own is not able to capture the interference behavior accurately enough [BF11].

In our approach, we need to utilize what information is available during the actual execution of an application and additionally keep the performance and memory overheads low. Our goal is to coarsely classify the applications in the noisy/quiet and sensitive/insensitive classes (two labels per application). We follow a similar path as in [HKG16, TMS11] where a larger number of hardware performance monitoring events is collected. We studied several such events provided by modern processors and their correlation with interference incidents. Clearly, applications that heavily utilize shared resources, such as caches and memory links, may potentially belong to the noisy and sensitive classes. Events that expose this kind of behavior are accesses, hits, and misses across the memory hierarchy, memory link bandwidth consumption, CPU stalls, and others. Interestingly, as also reported in [TMS11], sensitivity is much harder to predict than noisiness using standalone metrics, which is rather expected since characterizing a VM's behavior as noisy can be done by inspection, but characterization of sensitivity requires external probes which we do not use.

We built two machine-learning based classifiers, one for noisiness and one for sensitivity, using Support Vector Machines [CS00]. We used a rather small training set (140 labeled samples) and fed the two models with the following features:

- For the noisiness classifier we use LLC accesses per kilo instructions, LLC misses per kilo instructions, LLC miss rate.
- For the sensitivity classifier we use LLC accesses per kilo instructions, LLC misses per kilo instructions, DRAM bandwidth utilization, and total L2 pending miss stalls per total cycles.

We also use two intermediate classes (potentially noisy/sensitive) to capture symbiotic behavior that is not always clear, but ACTiManager acts pessimistically and also deals with potentially noisy and sensitive applications as they were pure noisy and sensitive. In future work we intend to assess if working with more fine-grained classifiers can lead to better overall results.

### Healthy state model

The interference model presented in the previous paragraph is utilized to avoid interference penalties when collocating VMs. However, for a number of reasons including the mix of the workload in a compute node, the misprediction of the model, or the unexpected change in the behavior of an application, unexpected interference may occur. In this case, ACTiManager takes action to mitigate this problem, and the first step towards this is to be able to recognize anomalies in the execution of an application.

While executing in the laboratory, we collect time series of performance monitoring events capable of describing the healthy state of an application. The key idea is that applications are potentially multi-phase, i.e., applications may exhibit different behavior during their execution lifetime, thus we record each of these phases that are observed under execution in a protected environment. We standardize the parameters of the collected time series and perform Principal Component Analysis (PCA) [KP01] in order to address the problem of potential variables that are highly correlated. Subsequently, we carry out data clustering, using either a linear model and K-Means, or a Gaussian mixture model and Expectation-maximization. The model, as well as the number of clusters, is determined by maximizing the silhouette score [R87] of the resulting clustered data. The limits within which the silhouette value of a data point is considered normal are delineated by a threshold factor. This factor denotes the standard deviation that the silhouette value can differ from its mean and it is adjusted so as to result in no more than 10% of the laboratory data being deemed anomalous. Any clusters that correspond to less than 5% of the data are removed from the model.

In the current implementation we have been relying on multiple monitoring events that we use to build the healthy state model, which are:

- branches, branch-misses, cycles, instructions, context-switches, cpu-migrations, page-faults, LLC-loads, LLC-load-misses, dTLB-loads, dTLB-load-misses, mem-loads, mem-stores.

In future work we intend to refine this set and reduce the number of events without trading-off accuracy. When the application is later on executed in a symbiotic environment, we compare the current state as built using the time-series of the events above with the model healthy state and report an issue if the current state deviates from the healthy one. In our approach, if the average silhouette value of the measurements exceeds a threshold limit, interference is assumed.

### 3.3.1.2 *Standard version of ACTiManager.internal*

ACTiManager.internal is responsible for enabling profitable consolidation within the node. To this direction, this component supports the following actions:

- mapping of virtual resources to physical resources, by pinning VMs on server CPUs and memory nodes in order to support resource optimization policies,
- monitoring the execution behavior of VMs and taking actions when performance anomaly is detected,
- migrating memory<sup>11</sup> from a NUMA socket to another for better locality and performance improvement.

ACTiManager.internal is executed periodically, which means that after each predefined interval, it searches for newly arrived VMs that are to be pinned accordingly to avoid interference, it checks for VMs that have completed their execution, and therefore need to be deleted from the system's representation, and finally it monitors all the VMs to check for anomalies in their execution in order to pin the affected VMs to different cores within the server to mitigate the interference.

Pinning of VMs is straightforward and achieved through the mechanisms provided by the hypervisor, i.e., through libvirt with KVM in our case. Of more interest is the decision of where to pin each VM considering the nature of workloads within the server. This decision is taken when a new VM is placed in the server by ACTiManager.external, when an anomaly is detected, or when an imbalance is detected at the termination of a VM's execution.

In a similar fashion, when it is deemed that the issue cannot be resolved by core pinning only, migration of memory content can be taken as an alternative action. Such alternative course of action can be taken, for example, when a new VM arrives and the current configuration is deemed nonoptimal, when two or more VMs that are antagonistic to each other share the same socket or memory bus, etc. The mechanism for memory content migration is provided by the hypervisor.

#### Pricing model

ACTiManager differentiates from previous research resource managers on the target function it applies to guide its decision making. Prior work [LCG15, YBM13] distinguishes only between latency critical and best effort applications, and targets first on the protection of the QoS of the latency critical applications and then the maximization of utilization with best effort applications. While we also distinguish between gold and silver applications based on their Service Level Objectives (SLOs) and pricing models, we utilize directly these pricing models and estimates of the slowdown of each application. In this way, we are more flexible in supporting further optimization policies beyond just performance, including the maximization of profit from the CSP perspective. Note that, under this scheme we are able to support decisions that could violate the QoS of a (either gold or silver) application, if that leads to higher profit, or decisions that better protect the QoS of the silver (best effort) applications that are being neglected currently in the literature. We incorporate these pricing models within the decision making process of ACTiManager.

---

<sup>11</sup> <http://man7.org/linux/man-pages/man8/migratepages.8.html>



### Gold VMs

We assume that gold services have SLAs with low tolerance to slowdowns compared to isolated execution, with the slowdown  $S$  defined as:

$$\text{Slowdown } S = \text{performance in isolation} / \text{performance attained}$$

Thus, the price  $m_g$  (in monetary units per time unit and per resource unit) for gold services is given by:

$$price_{gold} = m_g \text{ if } S \leq t_g$$

$$price_{gold} = 0 \text{ if } S > t_g$$

with  $t_g$  set to values close to 1 (e.g., 1.2 or 20% slowdown due to co-location/interference with respect to the performance achieved when running in isolation).

### Silver VMs

On the other hand, silver VMs may come in two different flavors. The first family requests cloud resources for a specific time period but poses loose restrictions on the quality up until a specific threshold. In this case, the pricing model is similar to that of gold VMs, but with a threshold  $t_s$  that takes much higher values (accounting for the possibility of silver VMs to sustain even higher slowdown due to interference with respect to the performance achieved when running in isolation):

$$price_{silver} = m_s \text{ if } S \leq t_s$$

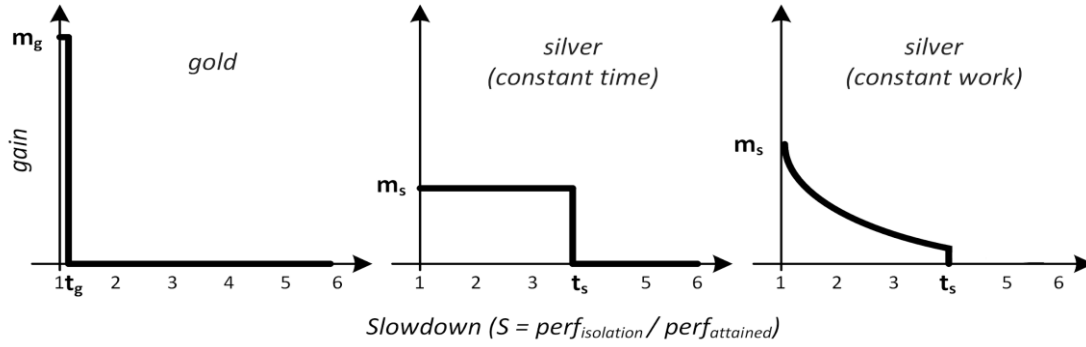
$$price_{silver} = 0 \text{ if } S > t_s$$

The second family of silver applications requests cloud resources to execute a specific job but with no strict demands for the time to completion (e.g., an analytics workload that takes approximately one hour, is submitted at midnight and should be finished before morning). In this case, the user expects to pay roughly a constant amount for the execution of her job. This is captured by a pricing formula as follows:

$$price_{silver} = m_s / S \text{ if } S \leq t_s$$

$$price_{silver} = 0 \text{ if } S > t_s$$

Figure 3.3 summarizes the three models that we use in our decision making within ACTiManager. Clearly, gold SLOs would be preferred by latency critical applications while silver SLOs would be preferred by best effort applications. We argue that the three discussed patterns are able to capture the needs of the large majority of cloud applications. Note also, that this modelling assumes that the Service Level Indicator (SLI) in our case is the slowdown compared to isolated execution and that there is a mechanism to calculate this slowdown in order to decide if and how much an application will be charged. We believe that this SLI is reasonable for an interference-aware resource manager, and can be easily computed using any metric required by the application to assess its own progress (e.g., throughput, tail latency) with straightforward monitoring mechanisms.



**Figure 3.3:** Pricing models for gold and silver applications.

We also consider that turning on a server in a cloud datacenter is associated with a specific monetary cost ( $m_{\text{server}}$  in monetary units per time unit) that is defined by the cloud provider and is related to the hardware capital (CAPEX) and operational (OPEX) expenditure of the datacenter. Depending on the power management technology of the server, this cost may be further broken down to the cost of turning on specific parts of the architecture. In our current implementation, we consider that there is a cost to turn on a NUMA socket of the server ( $m_{\text{socket}}$ ), but can be trivially extended to support more detailed power management schemes (e.g., turning on/off parts of the socket).

#### Placement model

ACTiManager.internal quantifies the profit from the CSP perspective of each candidate placement and picks the highest one. The profit is calculated with the aid of the aforementioned pricing models. Those models are fed with an estimated slowdown (ESD) and are aggregated over each application running on the server.

The ESD of a VM (vCPU)  $k$  is caused by core oversubscription (holds only for silver applications since gold applications are not oversubscribed) and interference due to the sharing of resources by other VMs running on the server. Thus, ESD is given by:

$$\text{ESD}(k) = S_{\text{oversub}} \cdot S_{\text{inter}}(k) , \text{ with}$$

$$S_{\text{oversub}} = 1 \text{ for gold applications}$$

$$S_{\text{oversub}} = \lceil S_{\text{vCPUs}} / (\text{cores} - g_{\text{vCPUs}}) \rceil \text{ for silver applications}$$

The placement rationale for ACTiManager.internal is the same as in the case of ACTiManager.external: the decision is taken utilizing an estimate of the slowdown (ESD) that will occur by placing a VM on the candidate locations within the server. The ESD for oversubscription and interference are refined as the internal component of the ACTiManager is fully aware of the location of each VM and can thus additionally utilize information on the hierarchical structure of the server HW and each application's CPU utilization. Thus, in this case we have:

$$S_{\text{oversub}}(k) = \max(1, \sum_{vm=1}^v \text{CPUUtil}_{vm})$$



and

$$S_{inter}(i, k) = \prod_{c=1, c \neq k}^{cores} \frac{\sum_{vm=1}^v S_{inter}(i, vm)^{\frac{1}{2^{distance(k,c)-1}}}}{v}$$

The slowdown due to interference imposed to VM<sub>i</sub> executing on core k ( $S_{inter}(i, k)$ ) is the product of the slowdowns caused by the workload executing in each core of the server but is properly adjusted (decayed) to their level of sharing resources in the server. To better capture such sharing levels, we construct a tree model of the server architecture (as described in Section 2), and we define the distance between two cores as the number of sharing levels that need to be traversed in the path between them. Clearly, the lower the distance, the higher the number of shared resources between the two cores, and thus the higher the impact on interference between different co-located applications. Figure 3.4 shows the representation of our tree model.

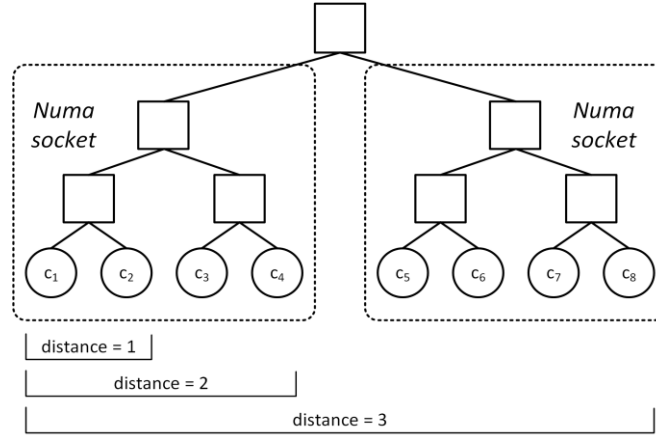


Figure 3.4: Server model.

ACTiManager.internal enforces the following strategy to map gold and silver VMs to physical cores.

For gold VMs it evaluates different alternatives for contiguous/consecutive cores that span the minimum levels in the server hierarchy, i.e., we assume that gold VMs benefit from being packed within the server. The approach utilizes the ESD presented before to select the best possible candidate. The process may involve placing elsewhere some silver VMs in order to make room for the gold VM that needs to be (re)placed.

For silver VMs the approach starts greedily to allocate each vCPU of the VM within the server again using the ESD discussed before to score all the alternatives. Note that, in any case the estimated profit of the new state after the new placement may be lower than the previous one. If this happens, ACTiManager.internal informs ACTiManager.external that this server is overloaded.

When an anomaly is detected, ACTiManager moves the suffering VM away from its current position within the node using the same aforementioned process. However, in order to mitigate the interference effects as much as possible, it disregards any costs incurred by switching on additional hardware components (e.g., a NUMA socket). Moreover, to avoid unnecessary and oscillatory

actions, ACTiManager moves a VM when performance anomaly is detected persistently for a predefined time window and avoids moving it again for another predefined time window. In case ACTiManager.internal detects that a recently moved VM is again suffering from interference, it notifies the ACTiManager.external that the server is overloaded. Note that alternatively ACTiManager could consider moving a different VM than the suffering VM, e.g., the smallest VM and/or low priority silver VMs. However, moving a different VM than the suffering VM does not necessarily mean that the performance of the suffering VM will improve, because there is uncertainty regarding which VM(s) exactly introduce interference. Instead, moving the suffering VM tries to directly mitigate performance interference.

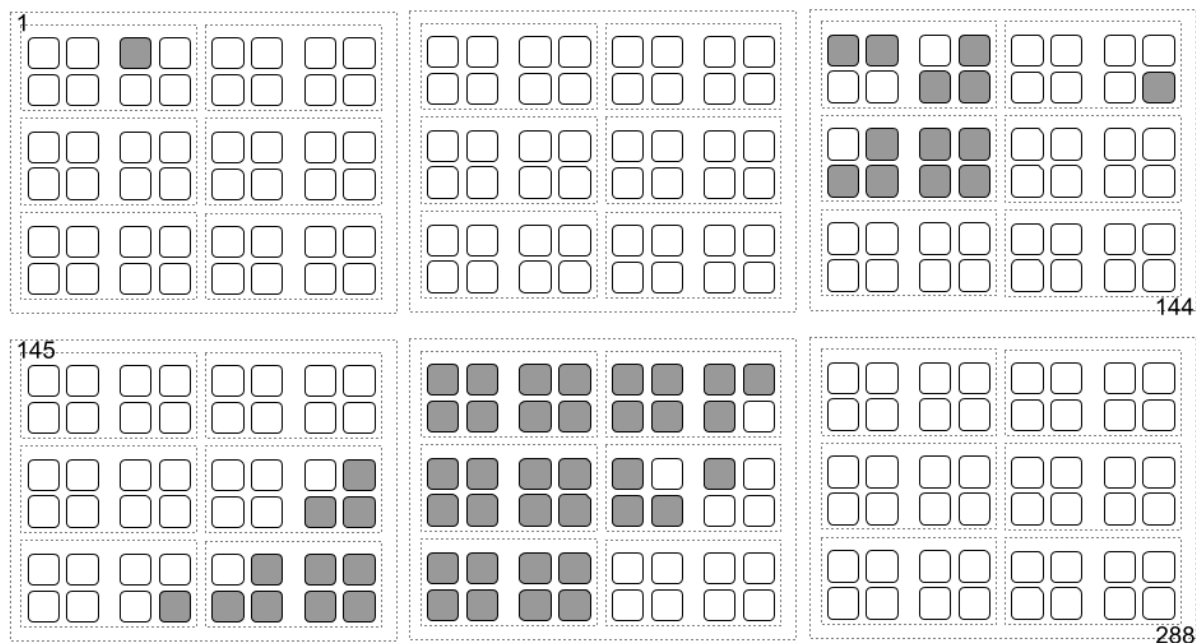
Finally, when a suffering VM is repinned to restore its performance after anomaly detection, ACTiManager.internal has to decide if it is necessary to move its corresponding memory to a different NUMA node depending on the destination set of cores. This decision is made by determining the NUMA node to which the majority of the destination cores belongs. Therefore, it is ensured that the majority of the VM's vCPUs have quick access to the application's memory.

#### *Placement model for large NUMA systems*

On large NUMA systems, such as the Numascale machines, multiple applications can be hosted in a single node making the mapping and performance resolution problem more complex. Besides interference among colocated applications, the performance of an application can be affected by how the resources composing the VM are aggregated, i.e., locality or remoteness — the distance between memory and cores.

Previous NUMA-aware works which discuss optimizing locality either consider small scale systems, i.e., a server with a few NUMA nodes, or use simulation to estimate the performance of applications on a large scale disaggregated system [BZF10]. A snapshot of the vCPU mappings during a run using the default scheduler used by KVM for applications that require resources beyond a single physical server under Numascale machine is shown in Figure 3.5. As shown in the figure, the resources are spread across different machines resulting in huge performance penalty due to cross server memory access. To make matters worse, we observe that the mapping changes during runtime due to variations in load and the inner workings of the scheduler.

To tackle performance issues due to resource spread out, we developed an additional placement algorithm for large NUMA systems that tries to place VM's cores close to VM's memory and performs contiguous allocation of resources for each VM giving preference to high priority application when it is not possible. Moreover, in such complex systems, unanticipated behaviours are bound to happen and when such behaviors happen, ACTiManager continuously remaps cores, and/or migrates memory in a less intrusive fashion.



**Figure 3.5:** Core mapping, Huge VM, as done by the default scheduler used in KVM.

Interestingly, applications when running in such systems behave differently and utilize the hardware in different ways depending on their characteristics. Some applications are very sensitive to interference, others are very sensitive to remoteness, while some others are sensitive to both. Classification of applications is a well-studied area [XL08, KBH08, KBH08] and the actual classification method used is independent of the resource mapping method. In fact, any method that correctly classifies applications into a few discrete categories depending on their cache usage and tolerance to cache sharing can be used with our method. For this component of ACTiManager we use the Animal classification scheme [XL08] to differentiate the degree of sensitivity for interference, as proposed by Xie et al. The Animal Classification scheme divides applications into different classes depending on their use of shared last-level cache. Applications are classified into the following animal classes:

- Sheep (quiet and insensitive): Gentle and tame applications that are not significantly affected by sharing cache with other applications.
- Rabbit (sensitive): applications that access caches fairly frequently and whose performance rapidly degrades with insufficient cache allocation or cache sharing with other applications.
- (Tasmanian) Devil (noisy and insensitive): Applications that access the cache very frequently, with high cache miss rates. As a result, they do not benefit much from caching and tend to negatively impact other applications as they "thrash" the cache.

The applications can be classified into the animal scheme, described above, by using either an offline or online method. In summary, our basic classification mechanism consists of running the candidate applications in a controlled environment with and without co-location with other workloads while measuring key performance indicators. In a production environment, the

classification can be performed by the application owner using similar techniques, or it may already be known by the developers.

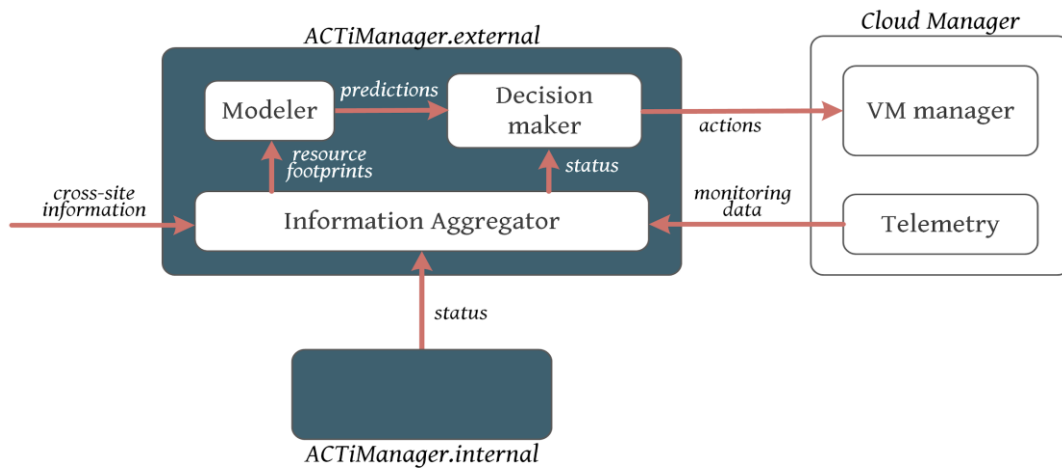
In addition to the animal classes, we also measure how an application is affected due to remote memory. We use two discrete classes, an application is either affected or unaffected. Combining these application characterization schemes, we can thus have applications, for example, "Affected Sheep" or an "Unaffected Sheep", etc. In scenarios where memory is scarce, for example, it makes sense to prioritize the use of local memory to sensitive applications, if both applications are gold applications.

Once we have the classification, we follow an algorithmic approach to map VMs into physical resources. A high-level description of the algorithm is described as follows. First, remoteness is addressed, that is applications larger than a single node and thus requiring remote memory are mapped. This is done when applications arrive in the system. An application should be "sliced" as little as possible, that is spread out over as few servers as possible. However, if an application is using a lot of RAM but fewer vCPUs than a single node, some of the memory on that node is reserved to run other, smaller, VMs on the remaining cores. The applications priority class is used when there is conflicting situation.

In the second stage, the algorithm tries to place the applications so as to cause as little interference between applications as possible. This can be performed using a combination of application runtime behaviour and high level objectives. Application runtime behaviour can be expressed either in terms of measured IP, MPI values or any other hardware performance counter. On the other hand, high-level objectives can be stated using the classification scheme, prioritisation and/or affinity values indicating which application can be co-located or not. For example, using the classification scheme, devil applications may not be placed with rabbits or other devils.

### 3.3.2 ACTiManager.external

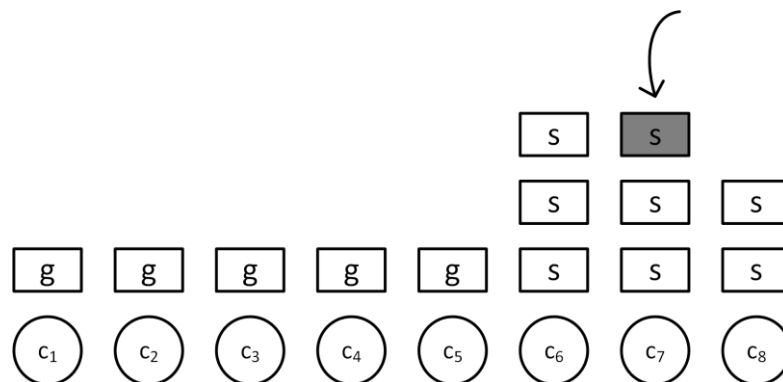
The operation and general architecture of the ACTiManager for the site level is shown in Figure 3.6. In this case, the Information Aggregator receives information: (i) from the Telemetry facilities regarding the resource utilization of the platform and the running VMs, (ii) from the various ACTiManager subcomponents of each node regarding more detailed information on the execution status of each node, and (iii) from peer ACTiManager components of remote sibling sites. This information is passed to the Modeler which, in this case, provides modeling and prediction information for cross-node and cross-site placement and migration actions in order to cope with problems like load imbalance between nodes, or site over/under-utilization. The Decision Maker then decides on more profitable workload allocations and requests the relevant actions from the core cloud manager. ACTiManager.external that handles cross-site offloading, called ACTiManager.multicloud, is described under Section 3.3.3.



**Figure 3.6:** General architecture and operation of the “ACTiManager.external” component that manages resources of a cloud site and across sites.

### 3.3.2.1 Allocating resources to servers

ACTiManager.external decides on which physical server to place a new VM after its classification in the "laboratory" node. To accomplish this, ACTiManager.external quantifies the profit from the CSP perspective of each candidate server and picks the highest one. The profit is calculated with the aid of the previously presented pricing models that are fed with an estimated slowdown (ESD) and aggregated over each application running on the server. Since ACTiManager.external is unaware of the exact placement of the VMs within the server, it makes a coarse assumption that considers even distribution of gold and silver VMs among the sockets of the server and non-oversubscribed gold applications. Figure 3.7 shows this assumption for an 8-core socket loaded with five gold vCPUs (gvCPUs), seven silver vCPUs (svCPUs) and one candidate silver vCPU to be potentially placed in this server.



**Figure 3.7:** Coarse assumption made by ACTiManager.external for vCPU placement within a server. g is used for gold applications and s for silver applications.

To simplify the discussion we consider single vCPU VMs. The ESD calculation is identical for gold multi-vCPU VMs, and can be easily extended to silver ones by progressively calculating the ESD of each vCPU to be placed in the server and finally keeping the maximum of ESDs of vCPUs to reflect

a pessimistic slowdown of the entire silver application. The ESD of a VM (vCPU)  $k$  is caused by core oversubscription (holds only for silver applications since gold applications are not oversubscribed) and interference due to the sharing of resources by other VMs running on the server. Thus, ESD is given by:

$$\text{ESD}(k) = S_{\text{oversub}} \cdot S_{\text{inter}}(k) , \text{ with}$$

$$S_{\text{oversub}} = 1 \text{ for gold applications}$$

$$S_{\text{oversub}} = \lceil S_{\text{vCPUs}} / (\text{cores} - g_{\text{vCPUs}}) \rceil \text{ for silver applications}$$

To provide an estimate for  $S_{\text{inter}}(k)$  for a VM  $k$  caused by all other neighbors in the system, we start from the estimate of the slowdown caused by application  $j$  ( $S_{\text{inter}}(k, j)$ ) which is calculated by the characteristics of the two VMs. More specifically, we use an estimated slowdown for the four combinations of noisy (n) / quiet (q) and sensitive (s) / insensitive (i) pairs, denoted as  $S_{\text{sn}}$ ,  $S_{\text{in}}$ ,  $S_{\text{sq}}$ , and  $S_{\text{iq}}$ .

We estimate  $S_{\text{sn}}$  as follows (the exact same procedure is followed for the rest of the three parameter combinations): we populate our server with a single vCPU of a sensitive application, fill the rest of the cores with noisy applications and repeat this experiment with a large number of representatives from both classes. We need an upper bound, pessimistic estimation, thus we keep the maximum slowdown  $S_{\text{max}}$  from this series of experiments after discarding outliers, and set

$$S_{\text{sn}} = S_{\text{max}} \wedge (1/\text{cores}-1)$$

The aggregate slowdown of VM  $k$  due to interference is then calculated as the superposition (product) of all other applications running on the system, distinguishing between gold and (potentially oversubscribed) silver applications, as follows:

$$S_{\text{inter}}(k) = \underbrace{\prod_{j=1}^{g_{\text{vCPUs}}} S_{\text{inter}}(k, j)}_{\text{interference from gold}} \cdot \underbrace{\left( \frac{\sum_{l=1}^{S_{\text{vCPUs}}} S_{\text{inter}}(k, l)}{S_{\text{vCPUs}}} \right)^e}_{\text{interference from silver}}$$

where

$$e = \min(S_{\text{vCPUs}} , \text{cores} - G_{\text{vCPUs}})$$

For the interference from silver applications we consider the average of their impact assuming the placement shown in Figure 3.7.

ACTiManager.external also monitors the status of all servers in the datacenter and takes actions to mitigate overload. If a number of servers complain, ACTiManager migrates excessive load to the less loaded servers gradually and taking care that oscillation will not occur. If the majority of servers complain, ACTiManager enters the “site overload” status and notifies ACTiManager.multicloud to resolve the situation. We integrate the above approach in Openstack’s Filter Scheduler.

### 3.3.3 ACTiManager.multicloud (Cross-Site Offloading)

ACTiManager.external manages the cloud within a single site and across multiple sites where cross-site offloading is responsible for the latter part. Even though it is part of ACTiCLOUD.external, it is put in its own section in order to clearly understand its architecture and how it works. The Cross-Site Offloading component, ACTiManager.multicloud henceforth, manages offloading of VMs between cloud sites. It is connected to multiple ACTiCLOUD deployment sites simultaneously and scales the number of VMs in each site through the OpenStack Heat API. ACTiManager.multicloud itself exposes an interface (see Section 6) which can be used by the ACTiManager.external that handles operations within a site to offload less prioritized VM instances to a different site when it no longer finds a local solution.

Figure 3.8 shows an overview of how the ACTiManager interacts with multiple ACTiCLOUD sites and exemplifies how a “silver” prioritized application with ingress requirements could be setup to support a multi-cloud environment by utilizing a local load-balancer and DNS. Note that in order to be a candidate for offloading, the application must be stateless as it is too expensive to undertake state migration across geographically distributed sites. The number of “silver” VMs could then be decreased and increased in both sites on-demand to make room for and support larger workloads in the local site.

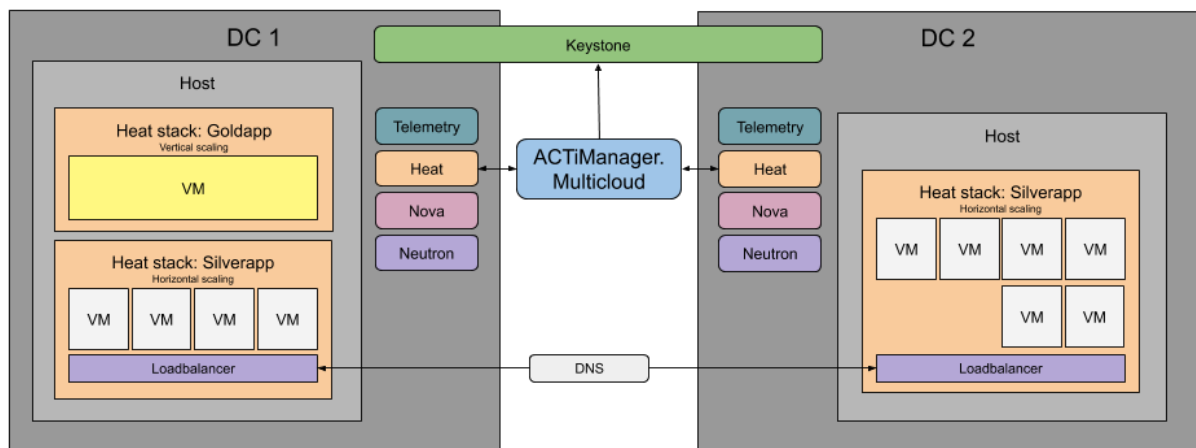


Figure 3.8: Interaction among multiple ACTiCLOUD sites.

### 3.3.4 ACTiManager - Cloud Manager roles and interaction

The ACTiManager and the cloud manager (OpenStack in our case) both lie at the same level of the base ACTiCLOUD architecture. Although OpenStack provides a rich set of capabilities to manage a cloud site, certain key features, such as interference-awareness, dynamic operation, prioritization, resource allocation optimization, and cross-site offloading, are missing. This is the part where the ACTiManager comes into play to interact with OpenStack and to provide additional functionality.

The core interaction between ACTiManager and OpenStack is shown in Figure 3.9. The Information Aggregator at both the node and site levels receives monitoring information from OpenStack’s Ceilometer. The Decision Maker at the site level (i.e., ACTiManager.external) requests from Nova to take actions on VM placement and migration, while the Decision Maker at the node level (i.e., ACTiManager.internal) requests from the hypervisor to take actions on VM co-scheduling and



placement, i.e., re-mapping virtual CPUs to physical CPUs or moving memory. Besides, additional features such as the possibility of creating and/or adding a VM into multiple instance groups (e.g., a VM can belong to either a "gold" or "silver" prioritization group based on the agreed SLA and billing policy, as well as to "noisy" or "sensitive" characterization group regarding the use of shared resources, based on its runtime behavior) are added inside OpenStack to augment both initial and continuous placement decisions. In general, ACTiManager.external should be viewed as a module that improves OpenStack by introducing entirely new components, adding new features to existing components of OpenStack, or using the APIs of OpenStack.

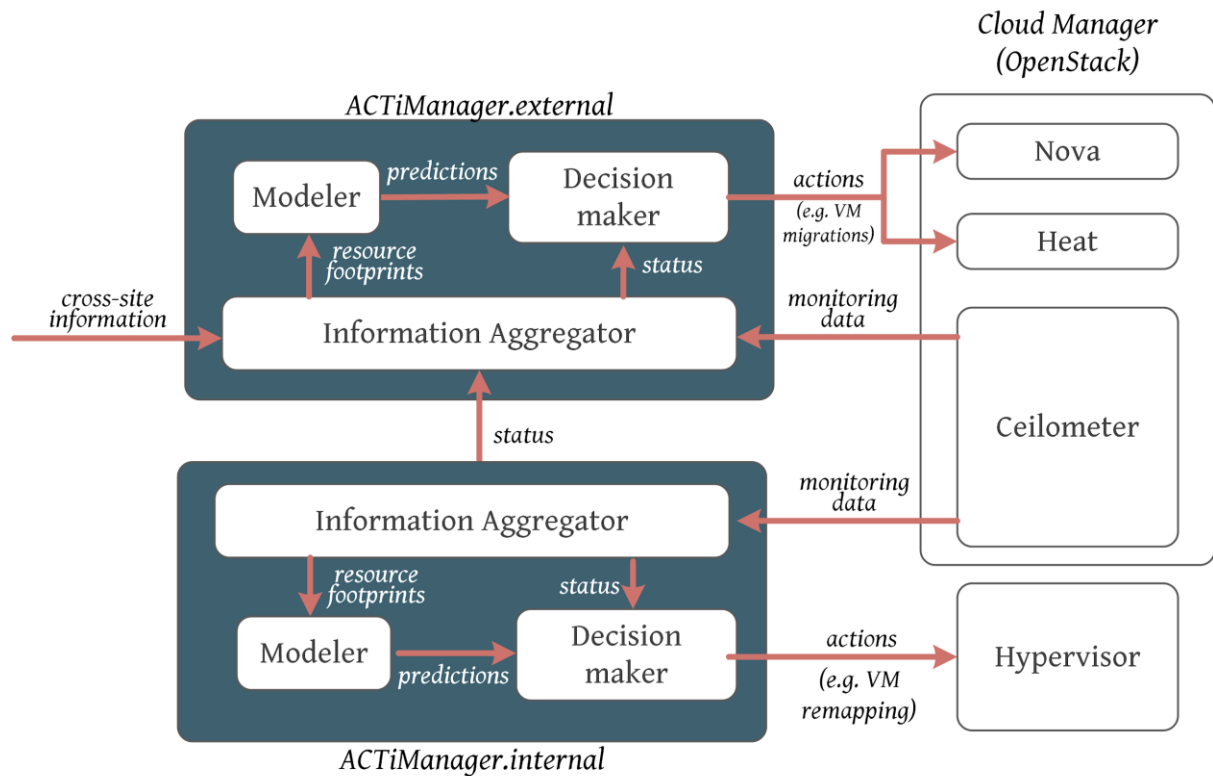


Figure 3.9: Interaction between ACTiManager and OpenStack.

### 3.4 ACTiManager Operation

Both the internal and external sub-modules of ACTiManager can be invoked periodically in specific Decision Periods (DP) that can be different for each sub-module. Decision Periods for both internal and external sub-modules are configurable values and are recommended to be set in the order of a few minutes, as ACTiManager needs to track longer-term behavior and gather monitoring data over sufficiently large periods to avoid temporary spikes. At the end of each Decision Period, ACTiManager analyzes the status of the system, decides on any identified event based on models, and performs any required actions. Next we describe these events, actions, and models that define the operation of ACTiManager.



### 3.5 Events

Events that trigger the decision making and actuation logic in the internal and external components of ACTiManager include:

- ACTiManager.internal events:
  - Node's total execution load changed: A new VM has been placed in that node or a VM has finished its execution.
  - Interference detected: One or more of the VMs that run in the node, experience performance degradation due to interference in shared resources (e.g., cores, caches, memory links).
  - Node under/over-utilization detected: Node resources are under/over-utilized, e.g., a low or high percentage of the node's processing cores or memory are used.
- ACTiManager.external events:
  - Creation of a new VM or re-characterization of an already characterized VM due to behavior change: A newly created VM is placed in the "laboratory" node for characterization in order to extract its "fingerprint" regarding its performance and behavior, i.e., CPU and memory utilization, IPC, IOPS, cache misses, and to classify it as a "noisy/quiet" or "sensitive/insensitive" VM regarding its use of resources. In addition, a VM is placed in the "laboratory" node when ACTiManager observes changes in its behavior and/or performance. Alternatively, the cloud user can directly provide this fingerprint information to ACTiManager (similarly to as providing the application's metric of interest); in that case, the newly created VM is managed directly by the "placement of a VM" event.
  - Placement of a VM (end of characterization period): A VM that was in a "laboratory" node until now, has just been characterized and needs to be placed in the production system. In case the cloud user provides directly the VM's fingerprint information, the same event takes place just when that VM is created.
  - Node(s) interference: One or more nodes report interference problems that cannot be mitigated at the node level by the ACTiManager.internal.
  - Site imbalance: Some production compute nodes are heavily utilized, while others are not (e.g., the difference between the node with the heaviest workload compared to the node with the lightest workload exceeds a threshold).
  - Site Overload: Site resources are over-utilized resulting in application demand not being met. To circumvent such situations some VMs are offloaded to different sister sites.
  - Notification from remote sites: One or more of the sibling remote sites report under/over-utilization.

### 3.6 Actions

On the occurrence of the aforementioned events, ACTiManager is able to select from a palette of actions that include the following actions:

- ACTiManager.internal actions:
  - VM (re)mapping: A VM's virtual CPUs and memory are re-mapped within the node. This responds to map a newcomer VM or to remap a VM that suffers from or causes interference to a more isolated place in the node (e.g., in a different CPU socket or NUMA node).
  - Report interference: ACTiManager.internal turns to the external component for interference resolution, in case internal actions (through remapping) are not effective.
  - Report under/over-utilization: ACTiManager.internal reports to the external component whether the node experiences under/over-utilization of resources.
- ACTiManager.external actions:
  - VM placement/migration: One or more VMs are migrated within the same site as a response to: (i) end of characterization period for that VM (see below), (ii) node interference, (iii) node over/under-utilization (e.g., need for load balance or for evacuation of node and shutting down),
  - VM termination: A VM is terminated/destroyed. One or more VMs are migrated within the same site as a response to the released resources.
  - Offloading to different sites: ACTiManager offloads some VMs to sibling site(s) to circumvent site overload.

### 3.7 Models

Deciding on which action to select under the occurrence of one or more events is the responsibility of the Decision maker. Decision making and event invocation are supported by the Modeling subcomponent. A list of the models is presented below:

- ACTiManager.internal models:
  - Characterization/classification model: This model extracts a VM's fingerprint regarding its use of resources, e.g., cores, memory, etc., and classifies its execution to either "noisy/quiet" or "sensitive/insensitive". This model is the key ingredient of the characterization phase that occurs while the VM is executed in the "laboratory" node.
  - Interference model: This model extracts the healthy state of the execution of a VM in an isolated environment free from interference, to dynamically identify it as either "interference suffering" or "interference free" when co-located with other VMs.

- Placement model: The model that decides in which physical CPUs and memory modules to place a new VM, taking into account the characteristics of the new VM and the characteristics of the currently running VMs of that node, using the estimated slowdown model.
- (Re)mapping cost model: This model predicts the performance of a VM for a certain mapping of its resources in the node, taking into account the characteristics of the VMs that currently run in that node. It also predicts the time required to remap a VM (i.e., its cores, memory, both) within the node, and predicts the new performance after the remapping action.
- Node under/over-utilization model: This model detects when a node is under/over-utilized.
- ACTiManager.external models:
  - VM placement model: The model that decides in which node to place a new VM, taking into account the characteristics of the currently running VMs in all or some of the site's nodes.
  - Site imbalance model: The model that detects when a site is unbalanced regarding the utilization of its nodes.
  - Site overload model: The model that detects when a site is overloaded.
  - Migration cost model: The model that predicts the time, bandwidth requirements, and failure probability to migrate a VM.

### 3.8 The Lifecycle of a VM under ACTiManager

Figure 3.10 summarizes the lifecycle of a VM within an ACTiManager-enabled system. ACTiManager heavily relies on online monitoring and analysis of a VM's "health" status.

To be able to detect any anomalies and take corrective actions, the system needs to have a solid understanding of VMs' execution characteristics, involving their performance, resource demands, required QoS levels, potential to suffer from or create interference, etc.

To accomplish this, every newly spawned VM starts its execution in the "laboratory" node, i.e., a node free from interference. During its execution in the "laboratory" node, ACTiManager collects monitoring information and, based on the characterization model, it creates the fingerprint of the VM, e.g., "noisy" or "sensitive" regarding resource utilization. The VM is executed and analyzed in the "laboratory" node for a specific Characterization period. Note that with this placement, the VM starts its execution normally.

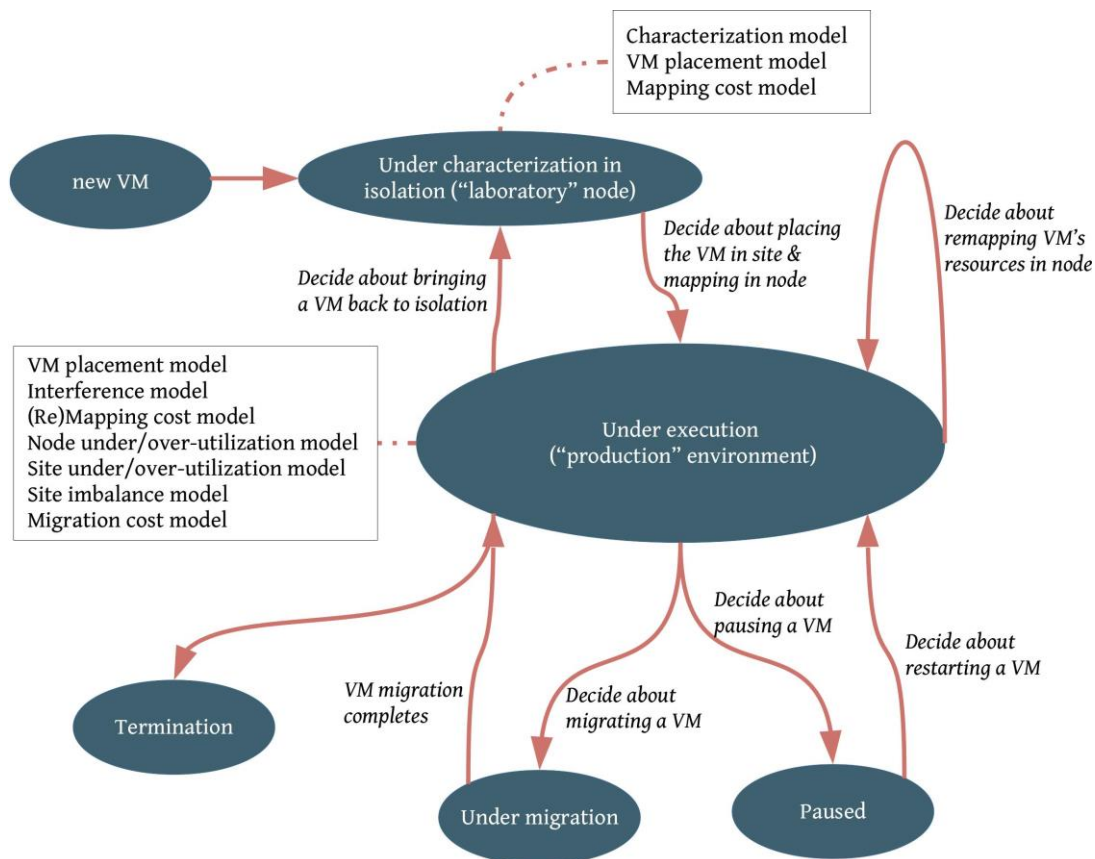
Upon the end of this Characterization period, ACTiManager.internal informs ACTiManager.external that in turn decides to place (migrate) the VM in its host node, where it is potentially co-executed with other VMs in a symbiotic environment. More specifically, ACTiManager decides the node in which the VM will continue its execution in the production environment, by checking if the resources that the VM requires are available in any node (number of cores, memory, etc.) taking into account the priorities (gold or silver) and the characteristics (fingerprint, noisy, sensitive) of the rest of the VMs/applications that run currently in the site's

nodes. Then, ACTiManager.internal (standard version) pins the newly arrived VM to the appropriate cores of the system and starts monitoring its behavior.

ACTiManager.internal and ACTiManager.external wake up in specific decision periods (that can be different for the two components) and check for various events, such as change in execution load, interference detection, and over-utilization, among others. In case none of the above events occur, ACTiManager keeps the current configuration and waits for the next decision period.

In case any of the above events occurs, ACTiManager decides how to resolve that issue based on the various models and by applying actions at node or site level. More specifically, ACTiManager.internal attempts to place the VM in different cores within the system. If unsuccessful, ACTiManager.external is informed that a server overload has occurred and migrates the VM in a different node.

Finally, in case the behavior of a VM changes significantly (e.g., change in fingerprint or performance), ACTiManager may migrate that VM to the "laboratory" node to perform another characterization phase and extract a new fingerprint.



**Figure 3.10:** The lifecycle of a VM under ACTiManager. The blue boxes represent the different states that a VM can be in, while the white boxes denote the models that are used when transitioning from the current state (denoted by the dotted line) to the next state.

## 4 ACTiManager invocation scenarios

In this section we describe in more detail the various invocation scenarios of ACTiManager. All these scenarios described below, except for the VM creation scenario, are initiated after a Decision Period quantum finishes; at that point, ACTiManager checks for any of the following scenarios and applies actions to mitigate them. In the VM creation scenario, ACTiManager is invoked every time a VM is created and destroyed.

### 4.1 The VM creation scenario

ACTiManager is invoked every time a VM is created. ACTiManager may place the newly created VM in the "laboratory" node for characterization and for extracting its fingerprint information. Once the characterization period for that VM finishes, ACTiManager decides where to place it in the production nodes of the site, taking into account the characteristics of the VM to-be-placed and the rest of the VMs in the nodes, in order to avoid interference and respect the SLA of the high-priority gold VMs. Note that ACTiManager may: (i) skip the characterization step when the cloud user provides the fingerprint information or when such information from previous runs of a user's instances are already generated to relax the need for characterizing all VMs, and (ii) bring a long-running VM back to the "laboratory" node for re-characterization, when the fingerprint and/or performance of that VM exhibits unpredictable behavior.

### 4.2 The interference detection scenario

ACTiManager detects any performance interference cases, after a Decision Period quantum expires. The identification of performance interference is part of the ACTiManager.internal's responsibilities.

ACTiManager.internal identifies interference based on the fingerprint of the VM. Once ACTiManager.internal identifies performance interference, it updates a per-VM counter which indicates the number of consecutive Decision Periods in which the VM in question has suffered from interference. When this counter reaches a predefined limit (default value is 5) ACTiManager.internal intervenes to alleviate the interference incident. The counter serves the purpose of reducing the number of actions taken by ACTiManager.internal by omitting the cases of momentary fluctuations in performance of the suffering VM or quick bursts of resource utilization by another VM which does not pose a long-term decline in the VM's performance.

ACTiManager.internal first tries to resolve interference locally at node level through VM remapping. In case the actions of ACTiManager.internal are not sufficient, which means that the remapped VM's performance is still detected to be inadequate just after the remap, ACTiManager.internal notifies the ACTiManager.external about the interference issue, to apply mitigation actions at site level. Indeed, ACTiManager.external can decide to migrate a VM to a different compute node. ACTiManager.external could decide among different policies. For example, ACTiManager.external could migrate either one of the "sensitive" VMs that suffer from interference, the "noisy" VM that causes interference, the smallest or the largest VM in terms of resources (e.g., CPUs, memory, storage). The current version of ACTiManager migrates the "sensitive" VM that suffers from interference; however, ACTiManager could be easily extended to implement the other policies.

Another available action could be that of pausing; assuming the scenario of having a gold and a silver VM running in the same node and the gold VM experiencing performance degradation due to interference, ACTiManager.external could decide to pause the execution of the silver VM (which is typically assumed as a VM that runs batch applications) and restart it again later, in order to improve the performance of the gold VM and meet its QoS level for as long as it is deemed necessary. Another option could be to re-characterize the "noisy" or the "sensitive" VM in order to better understand whether this behavior occurs due to some phase change in its execution. The latter two options are not currently implemented in the v2.0 of ACTiManager; however, they could be easily added in the palette of ACTiManager's actions.

### 4.3 The overload, underload, and fragmentation detection scenarios

ACTiManager detects system overload, underload, or fragmentation, after a Decision Period quantum expires. These events entail different courses of actions. ACTiManager collects statistics about the execution of the VMs in the nodes and the node utilization, and uses threshold-based algorithms to detect underload and overload.

ACTiManager detects overload at server level by estimating the current potential gain through the estimated slowdown (ESD) and comparing that with the value of the previous Decision Period. In case the current potential gain is lower than previously, ACTiManager.internal notifies the external component and suggests a VM to be migrated in order to mitigate the server overload.

ACTiManager.internal suggests a VM that is executed in the corresponding server by examining its VMs and choosing from the silver and noisy VMs, those that have the most vCPUs. In case there are no silver or noisy VMs, a gold or quiet (accordingly) VM will be chosen. In case of a tie between the largest VMs (in terms of vCPUs), the VM with the vCPU that has the maximum estimated slowdown will be chosen.

In each Decision Period, ACTiManager.external receives the server overload notifications from each internal module. Afterwards, it calculates the percentage of the internals that have reported overload. In case that the computed percentage is less than or equal to a predefined limit (set by default at 25%), ACTiManager.external handles all the requests by migrating the suggested VMs to the least loaded server (in terms of assigned vCPUs, on tie the server with the least gold vCPUs is chosen). Otherwise, if the percentage is greater than the aforementioned limit but less than a second limit set to 50% by default, ACTiManager.external handles half the requests. In case the percentage is greater than the second limit, no action is taken. This prevents oscillating scenarios from happening, where handling a server overload notification of one server, thus migrating a VM to another server, leads the destination server to an overload state.

The thresholds are periodically checked and events are generated when they are violated so that the Decision Maker can take corrective actions. For example, when the system is determined underloaded, that is an indicator to either admit more VMs to maximize revenue, or to consolidate the VMs to minimize energy cost. On the other hand, detecting system overload is an indicator for VM migration, pause, or termination to satisfy demands of high priority applications.

Fragmentation detection is also performed by checking the allocated resource distribution for each VM. The decision maker tries to compose resources for a VM from a nearby node as much as



possible in NUMA architectures, as is the case with the Numascale platform. This resource distribution is periodically adjusted depending on the behavior from neighbors VMs.

#### **4.4 Imbalance detection scenario**

ACTiManager is invoked periodically in specific Decision Periods (DP) that can be different for the internal and external components, to check for load imbalance among the site's nodes. For every Decision Period, ACTiManager.internal checks the utilization of the resources at node level, and in case of under-/over-utilization, it reports that event to the ACTiManager.external. In this way, ACTiManager.external has complete view of all nodes' utilization, and can identify whether load imbalance among the site's compute nodes exists. In case load imbalance is detected, ACTiManager.external can perform VM migration actions based on the available models to balance the work among the site's compute nodes, while taking into consideration the necessity of avoiding performance interference. In addition, load imbalance can be detected when a newly created VM has been characterized and is about to be placed in the production system, or when a VM is destroyed.

#### **4.5 Application under performance**

While under performance of application is determined and managed by the cloud user or application owner, application owner can report performance issues to ACTiManager as a hint which can be used during decision making. The performance agent is used for such purpose, as described in Section 5.

#### **4.6 Cross-site offloading scenario**

In the case where an issue cannot be resolved at site level, the ACTiManager can utilize remote sites by offloading VM(s) as a last resort. This is done using a scaling controller which distributes a set of VM(s) across multiple sites. Commonly these VM(s) would be of silver priority or batch jobs as shared state across large geographical distances is less feasible than inside a single region. Whenever the ACTiManager determines it is necessary to utilize the cross-site functionality a reconfiguration of the scaling weight for each site is done. The weights are translated into the number of VM(s) needed to run depending on the total number of VM(s) configured for the VM set. The resulting behavior would be either a scale-out or scale-down in the different sites. The local sites then control the placement like it does in the common creation scenario.

## 5 Module description

In this section we describe in detail the software components of ACTiManager that have been implemented in this deliverable (ACTiManager v2.0).

Our implementation and final release of ACTiManager focused on supporting the entire lifecycle of a VM, as described in Section 3. Towards that goal, we created and placed accordingly all the functional and modeling components that are required to support the design of ACTiManager and the invocation scenarios of ACTiManager, as described in Section 4, and integrated them with OpenStack.

Our implementation of ACTiManager assumes commodity, off-the-shelf, hardware platforms (Intel/AMD) and hypervisor technology (QEMU/KVM<sup>12</sup>), in order to avoid any dependencies that could arise during the integration of the MicroVisor (the hypervisor technology that is provided by ONAPP) with the ACTiCLOUD's hardware platforms, i.e., the Numascale and KMAX platforms (that are provided by NSCALE and KALEAO, respectively). Hence, our implementation is independent of the underlying hardware, it only depends on the hypervisor and its integration with OpenStack, and consequently can be used on the Numascale and KMAX platforms with KVM as hypervisor.

### 5.1 Overview

Our final version of ACTiManager implementation consists of the two fundamental sub-components of ACTiManager: (a) ACTiManager.internal and (b) ACTiManager.external. ACTiManager is primarily developed as a separate component that utilizes and complements an OpenStack installation and its core components. The current version of ACTiManager is based on the Pike<sup>13</sup> version of OpenStack. As OpenStack is implemented in Python, we decided to implement ACTiManager in Python, too. In addition, both ACTiManager.internal and ACTiManager.external sub-modules consist of three separate components: (i) the information aggregator, (ii) the modeler, and (iii) the decision maker. These components are responsible for different tasks at the two levels and implement different functionality.

### 5.2 Internal and External sub-modules

#### 5.2.1 Internal

ACTiManager.internal sub-module is implemented as a daemon service and consists of the laboratory version and the standard version. It executes in a continuous loop in which it sleeps for a predefined time, named as the Decision Period (DP). Then, it wakes up and evaluates the current state of the node and takes the appropriate decision at node level. More specifically, it gathers monitoring information through its Information Aggregator component and checks via its Modeler component for the existence of the following conditions:

- whether total execution workload has changed compared to the previous Decision Period,

---

<sup>12</sup> [https://www.linux-kvm.org/page/Main\\_Page](https://www.linux-kvm.org/page/Main_Page)

<sup>13</sup> <https://www.openstack.org/software/pike/>



- whether node under/over-utilization exists,
- whether performance interference exists.

Based on these conditions, ACTiManager.internal proceeds to the corresponding decision. The main actions that the internal sub-module is capable of applying in this version of ACTiManager are:

- to (re)map a VM's virtual CPUs and memory within the cores and the NUMA nodes of the server,
- to report interference to the external sub-module via the communication mechanism of RabbitMQ that is described next,
- to report node under/over-utilization to the external sub-module via the communication mechanism.

### 5.2.2 External

ACTiManager.external sub-module is also implemented as a daemon service. It executes in a continuous loop in which it sleeps for a predefined time, named as Decision Period (DP). Then, it wakes up and evaluates the current state of the site, taking into consideration the state of all individual nodes, to apply the appropriate decision at site level and between sites during cross-site offloading when a site is overloaded. More specifically, it gathers monitoring information through the Information Aggregator component (and the internal sub-module) and checks via the Modeler component for the existence of the following conditions:

- whether the characterization period for those VM(s), if any, that are in the "laboratory" node(s) has been completed,
- whether site imbalance exists,
- whether site under/over-utilization occurs,
- whether there are any messages in the communication mechanism between the external and internal sub-modules for interference detection (the internal sub-module notifies the external).

Based on these conditions, ACTiManager.external undertakes decisions accordingly. The main actions that the external sub-module is capable of applying in this version of ACTiManager are:

- to migrate one or more VMs from one node to another,
- to pause one or more VMs,
- to offload one or more VMs from overloaded site to less overloaded site.

### 5.2.3 Communication mechanism between Internal and External sub-modules

The internal and external sub-modules need to exchange information. For example, ACTiManager.internal needs to post important events, that it cannot handle itself, to the external sub-module, which in turn applies the corresponding high-level action. This communication

occurs via the mechanism that has been implemented using RabbitMQ<sup>14</sup>. RabbitMQ is an open source message broker software that originally implemented the Advanced Message Queuing Protocol (AMQP) and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol (STOMP), Message Queuing Telemetry Transport (MQTT), and other protocols. The events of communication between the internal and external sub-modules are:

- periodic update of node monitoring information,
- detection of node under/over-utilization, and
- detection of interference between VMs in the same node.

### 5.2.4 Communication support between ACTiManager.internal and hypervisor

The internal sub-module communicates with the node's hypervisor using a wrapper library, which provides a hypervisor-agnostic API. This library essentially provides an interface, which in turn is implemented by hypervisor-specific components. This library augments the Modeler and the Decision Maker, via the Information Aggregator, with detailed performance statistics, and enables the Decision Maker to apply the (re)mapping and capping actions.

We have implemented a "libvirt-specific helper library" that provides primarily KVM-specific operations. As libvirt supports also the Xen hypervisor, this library can be extended to support Xen as well. The libvirt helper library provides the internal sub-module with the ability to apply actions at node level and gather system-level topology and monitoring information that is currently unavailable on Openstack. More specifically, the helper library provides the ability to:

- query and change dynamically the mapping of virtual CPUs (vCPUs) to physical CPUs (pCPUs),
- query the node's NUMA information and topology,
- tune the NUMA memory options for the VMs,
- get fine-grained node and VM statistics (CPU and memory statistics, measurements using performance counters, etc.).

Note that those operations that are already supported by OpenStack are omitted from these components, such as VM (live) migration, coarse-grained monitoring information and statistics about the VMs and the node, etc.

## 5.3 Information Aggregator Component

### 5.3.1 Internal

The Information aggregator component of the internal sub-module is responsible for gathering monitoring information at the node level. More specifically, it maintains or retrieves information metrics about:

---

<sup>14</sup> <https://www.rabbitmq.com/>

- the topology of the node, e.g., number of physical CPUs, amount of RAM, NUMA configuration,
- the mapping of virtual CPUs to physical CPUs,
- runtime execution metrics such as CPU utilization about the VMs,
- runtime hardware performance counter values for each VM, such as IPC, MPKI.

### 5.3.2 External

The Information aggregator component of the external sub-module is responsible for the high-level view of the site infrastructure. More specifically, it retrieves information metrics about:

- the total number of active nodes,
- the total number of running VMs,
- the number of running VMs per node,
- the characteristics of the running VMs (gold, silver, noisy, sensitive),
- the number of paused VMs, and
- the number of characterized VMs that need to be placed.

### 5.3.3 Performance Agent

The Performance Agent is a lightweight component that runs in the host VM and exposes a rest interface, used by the host applications to:

- Set performance alerts
- Reset performance alerts

The Performance Agent gathers status information from applications that indicate whether the applications are performing as well or not. An analysis of the Performance Agent is provided in Section 6.6.

## 5.4 Modeler Component

### 5.4.1 Internal

The Modeler component of the internal sub-module is responsible for the following functions:

#### *VM characterization model*

This model extracts a VM's fingerprint and classifies it regarding its use of resources, e.g., cores, memory, etc., and classifies its execution to either "noisy" or "sensitive". This model is the core of the characterization phase that occurs while the VM is executed in the "laboratory" node. Section 3 describes this model in detail.

#### *Anomaly/Interference detection model*

This model extracts the "healthy state" model to later identify whether its execution suffers from interference or not when co-located with other VMs. Section 3 describes this model in detail.

### *Estimated Slowdown model for placement*

This model targets generic NUMA servers and estimates the slowdown that the VMs may experience due to co-location by taking into consideration the characteristics of all VMs running on the server. Section 3 describes this model in detail.

### *Estimated Slowdown model for large NUMA systems*

This model targets particularly large NUMA servers, such as the Numascale hardware platform, and estimates the slowdown that the VMs may experience due to co-location or remoteness by taking into consideration the characteristics of all VMs running on the server. Section 3 describes this model in detail.

### *Remapping cost model*

This model predicts the time required to remap a VM (cores, memory, both) within the node, and the expected performance improvement/degradation after the remapping action. This model targets two main characteristics that are present in typical servers, i.e., NUMA and core heterogeneity. Note that these two characteristics are exemplified by the project's two hardware platforms, i.e., the NUMA architecture of the Numascale system, and the heterogeneous big.LITTLE cores of the Exynos boards that constitute the KMAX system. Thus, we split this model into two parts:

- Remapping cores and memory in a NUMA system. The Numascale platform consists of multiple servers that implement the NUMA architecture themselves. As the Numascale platform introduces an additional level of NUMA shared-memory across multiple servers, the mapping of VM's cores and memory becomes an important factor in defining its performance in such systems. This model focuses on the cost of (re)mapping the memory of a running VM.

More specifically, the model predicts the performance of the VM under the new configuration, in case a part or the whole memory of a running VM is remapped on a different NUMA node away from the assigned cores. Such actions could be the result of ACTiManager's decision making; for example, in order to create free memory on one or more NUMA nodes to accommodate the creation of a new large VM, ACTiManager may remap the memory of running VMs. Based on this model, ACTiManager can select the VM that will suffer less from performance degradation due to accessing remote NUMA nodes. In addition, the model predicts how much time it would take to remap the memory from one NUMA node to another.

We followed a machine-learning approach to predict the impact on performance of core and memory placement in non-uniform memory access (NUMA) systems<sup>15</sup> [AKP18]. The impact on performance depends on the architecture and the application's characteristics. We focused our study on features that can be easily extracted using common hardware

---

<sup>15</sup> Fanourios Arapidis, Vasileios Karakostas, Nikela Papadopoulou, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. "Performance Prediction of NUMA Placement: A Machine-Learning Approach". In 1st International Workshop on Next Generation Clouds for Extreme Data (XtremeCLOUD 2018) - held in conjunction with CloudCom 2018.

performance counters, i.e., misses that occur in the last-level cache (LLC) and the TLB, and instructions per cycle (IPC). We use benchmarking data from a wide set of single-threaded benchmarks from Spec2006 and Parsec, to train multiple regression models that could serve as performance predictors. Our experimental results show notable accuracy in predicting the impact on performance, with relatively simple, yet non-linear, prediction models.

To predict how much time it would take to remap the memory of a running VM from one NUMA node to another, we measured the remapping time for micro-benchmarks and real world applications with various working sets, and devised a simple linear model that is primarily affected by the characteristics of the underlying architecture, i.e., the larger the memory footprint is, the longer the time is needed to remap the memory of an application from one NUMA node to another.

Note that our current implementation of ACTiManager follows a simpler policy that moves the memory of the VM accordingly so that the majority of vCPUs are on the same NUMA node with the VM's memory. However, that simple model can be extended with the aforementioned one.

- Remapping cores in big.LITTLE systems. The KMAX platform consists of multiple Exynos boards, with each board implementing ARM's "big.LITTLE" architecture. big.LITTLE is a heterogeneous computing architecture, that couples relatively more powerful and power-hungry processor cores (big) with relatively battery-saving and slower ones (LITTLE). The execution of a VM on a big/LITTLE core may have significant impact on its performance. However, the impact on performance is not the same for all applications, and it depends on the application itself. Hence, the mapping of VMs on cores is important.

An intuitive approach for mapping VMs to cores is to prioritize the gold VMs by mapping them on the big cores, while mapping the silver VMs on the LITTLE cores. Our current implementation of ACTiManager follows that approach.

In addition to that approach, we experimented with performance modelling that targets the scenario of mapping multiple VMs to big/LITTLE cores when they belong on the same prioritization class, based on a different approach. The idea is to use various metrics from the footprint of the VM and low-level monitoring events from performance counters, to predict the impact on performance when remapping a VM from a big to a LITTLE core, and vice versa. More specifically, we performed experiments with various benchmarks and collected statistics regarding the instruction mix, the cache behavior, the performance and the energy consumption when running on big and LITTLE cores. Then we used various machine learning models, both linear and nonlinear, to predict the impact on performance when moving from one core type to another and enable better placement policies.

#### 5.4.2 External

The Modeler component of the external sub-module is responsible for the following functions:

##### *Site Under/Overutilization model*

This model detects when a site is under/over-utilized. For the detection of this scenario, a minimum and a maximum number of VMs per node are used, together with the characteristics of the VMs. In case the total number of VMs is less than the minimum number multiplied by the total number of nodes, i.e., the ratio of VMs per node is low, then a Site Underutilization event is raised. Similarly, in case the total number of VMs is more than the maximum number multiplied by the total number of nodes, i.e., the ratio of VMs per node is high, then a Site Overutilization event is raised.

##### *Imbalance detection model*

This model detects when a site is unbalanced regarding the utilization of its nodes in a coarse-grained manner. More specifically, we use a maximum difference threshold regarding the number of VMs that are executed on the node with the heaviest load compared to the node with the lightest load. In case this limit has been reached or exceeded, then an Imbalance event is raised.

##### *Migration prediction model*

This model is responsible for estimating the migration cost of a VM from one node to another. Our ongoing effort on this model has primarily focused on live VM migration within QEMU/KVM and is based on previous research [ZDL17].

We have observed that the most significant parameter that affects the migration procedure is the number of dirty pages per iteration of the migration, i.e., the rate at which the VM writes memory affects the feasibility and the time needed for performing a VM migration action. We combine this dirty page write together with the total available network bandwidth to predict at a relatively high accuracy if the migration will succeed (we focus on the pre-copy algorithm<sup>16</sup>) and to estimate the total time of migration and the total downtime that the user will experience. Note that this model is not part of the current version of ACTiManager; instead, the current implementation uses simple threshold-based and linear models based on memory usage for predicting the feasibility and the cost of migration, respectively.

The biggest limitation of the pre-copy migration algorithm is that, under specific conditions, the migration process may not converge and thus never be completed. Specifically, this can happen if the dirty page rate of a VM is higher than the available network bandwidth. In order to get an estimation of the dirty page rate of a running VM, we designed and implemented BitmapTrace, a tool that leverages the `ioctl(KVM_GET_DIRTY_LOG)` system call exported by the KVM kernel module.

Considering the case of an overloaded physical node as part of a cloud infrastructure, a profound mitigation action is to rebalance the VMs running on this node and transfer them to less loaded nodes. In order to proceed with that, we envision a model that chooses the most appropriate VMs among a pool of possible candidate VMs for migration. We build this model having as input the

---

<sup>16</sup> [https://en.wikipedia.org/wiki/Live\\_migration](https://en.wikipedia.org/wiki/Live_migration)

dirty page behavior using the BitmapTrace tool and also the downtime-limit, which expresses the maximum tolerable downtime for a VM. Our model returns a list of VMs to be migrated in descending preference order taking into consideration the corresponding SLA of each VM.

## 5.5 Decision maker

### 5.5.1 Internal

The Decision Maker component of ACTiManager.internal proceeds to the corresponding decision based on the aggregated information and the output of the models. The main actions of the internal sub-module are to (re)map a VM's resources among the NUMA nodes of the node, and to report interference and node under/over-utilization to the external sub-module via the communication mechanism.

### 5.5.2 External

#### *Placement*

The Decision Maker component of ACTiManager.external considers various placement policies when placing a VM. The VMs priority class (e.g., gold, silver) and its runtime behavior (noisy, sensitive) are taken into account when a VM is placed in a node, together with the current load of the various nodes. Section 3 describes this model in detail. Our approach to implement the placement functionality of the Decision Maker is through modifying the OpenStack Filter Scheduler.

In addition, we support the Decision Maker component by interacting and modifying existing components of OpenStack. More specifically, we modify OpenStack to be able to create:

- Global instance groups, i.e., instance groups that are not scoped to a single project as in vanilla OpenStack, but rather allow for membership of instances from all the projects in a cloud.
- Multi-group instance memberships that allow instances to be added as a member to multiple groups, instead of just being member to only one group as in vanilla OpenStack. Each instance group is taken into consideration by the scheduler when the instance is being placed on a node.

These modifications allow us to create multiple groups with different policies (e.g., anti-affinity policy which prohibits VMs to be placed on the same node together with other VMs that belong in the same group) which in turn allows for multiple placement policies for a single VM. In this way, ACTiManager can group instances to guide the placement process in a more sophisticated way than was possible before in vanilla OpenStack. For example, instance A can be added to both a global anti-affinity group called "gold" and to another global anti-affinity group called "noisy", while instance B can be "gold" and "quiet". Based on this information, the scheduling policies can be modified to make more informed decisions when placing VMs.

## 5.6 Interaction with OpenStack

Our current version of ACTiManager is able to interact with OpenStack's nova-compute component through the NOVA API to:

- create and manage instances,
- get a list of current instances,
- live migrate instances,
- get different monitoring information

The OpenStack modifications of ACTiManager target both the CLI and Nova components to support the following functionalities:

- Global instance groups
- Multi-group membership
- Adding and removing instances from groups

Finally, the following features are implemented in OpenStack Heat<sup>17</sup>:

- multi-cloud operation such as creating and listing multicloud stack.

---

<sup>17</sup> <https://wiki.openstack.org/wiki/Heat>



## 6 Documentation / User manual

### 6.1 ACTiManager.internal

ACTiManager.internal is implemented in "Internal.py" and it runs as a daemon service in each node. It executes in a continuous loop in which it sleeps for a Decision Period time, named as Decision Period (DP). Then, it wakes up and evaluates the current state of the node and takes the appropriate decision at node level.

### 6.2 ACTiManager.external

ACTiManager.external is implemented in two parts. The first part is implemented as an Openstack filter scheduler's filter and is responsible for the initial decision of the compute node where a new VM is going to be placed. This part is triggered each time a new VM is created or when a VM is cold migrated (when using live migration the target compute node needs to be specifically specified and the Openstack filter scheduler is bypassed). The second part of ACTiManager.external is implemented in "External.py" and it runs as a daemon service for each site on the OpenStack controller node. The main responsibility of this second part is to listen for messages from the ACTiManager.internal daemons and act appropriately. It executes in a continuous loop in which it sleeps for a Decision Period time. Then, it wakes up and evaluates the current state of the node and takes the appropriate decision at site level, as described in Sections 3 and 4.

### 6.3 ACTiManager.multicloud (Cross-site Offloading)

The ACTiManager.multicloud cross-site controller is implemented as a standalone service which runs as a daemon. The user configures the cross-site daemon with the OpenStack credentials for each site it should control. The user then sets up each Heat stack which should be scaled by the cross-site service, either using the CLI tool or the REST API. Finally, the ACTiManager.external is configured with the cross-site daemon's API endpoint which it uses to reconfigure the site scaling weights.

Whenever the cross-site service sees a change in the site weights, it will scale-up or scale-down the number of VMs in each Heat stack. If the cross-site service is unable to do so, for example in the case of a site outage or a misconfiguration in one of the sites, the weight will be re-distributed on the other available sites.

Next a few basic commands follow that the user can use to manually setup and configure the ACTiManager.multicloud component using the CLI tool.

Start the controller daemon:

```
actimanager-multicloud run
```

Add a muticloud stack:

```
actimanager-multicloud add stack_name \  
    --count [initial desired total VM count] \  
    --parameter [Heat stack count parameter]
```

Add (or update) a cloud weight configuration to the multicloud stack:

```
actimanager-multicloud weight set stack_name cloud_name \
    --weight [amount of total count (from 0 to 1)]
```

List all configured multicloud stacks:

```
actimanager-multicloud list
```

```
+-----+-----+-----+-----+
| Stack name | Desired count | Count parameter | Clouds          |
+-----+-----+-----+-----+
| silverapp  | 5              | count           | athens, manchester |
+-----+-----+-----+-----+
```

Show details for a single multicloud stack:

```
actimanager-multicloud show
```

```
+-----+-----+
|      Attribute | Value          |
+-----+-----+
|      Name      | silverapp      |
| Desired count  | 5              |
| Count parameter | count          |
|      Weights   | athens (50.0%) |
|                | manchester (50.0%) |
+-----+-----+
```

## 6.4 Characterization agent

The characterization agent that runs on the "laboratory" node is implemented in the "characterization\_agent.py". The agent runs continuously in that node and checks whether there are any VMs that are waiting for characterization. If so, the characterization agent runs the VM in the "laboratory" node for a given period of time and extracts statistics, such as CPU utilization. Based on simple thresholds, the characterization agent identifies the VM as "noisy" or "sensitive" and marks the VM as characterized. Note that the list of the waiting VMs is updated by the ACTiManager.external.

## 6.5 Communication mechanism

The communication between the internal and external sub-modules is necessary for various purposes, as described in Sections 3, 4, and 5. The communication has been implemented via the RabbitMQ mechanism through the Pika driver.

## 6.6 Performance Agent

The ACTiManager Internal Performance Agent is a lightweight component that runs inside the guest VM. It is an optional component, and ACTiManager.internal will poll it at the startup of a new VM. If it responds, it will continue to poll it. The application running inside the host VM can

communicate with the Performance Agent through a REST interface. Using this interface a performance alert can be set or reset. When ACTiManager.internal polls the Performance Agent and finds that a performance alert has been set, it will know that the application is experiencing performance issues and can take appropriate action. When the situation has been resolved, the host application should reset the alert.

## 6.7 Libvirt

The internal sub-module communicates with the node's hypervisor using a wrapper library, which provides a hypervisor-agnostic API. This library is implemented in "libvirt\_helper.py". The libvirt helper library provides the internal sub-module with the ability to apply actions at node level and gather system-level topology and monitoring information that is currently unavailable on OpenStack. It currently provides primarily KVM-specific operations as described in Section 5.2.4.

## 6.8 OpenStack modifications for Placement

The placement functionality of the Decision Maker is implemented through interacting and modifying existing components of OpenStack. More specifically, we modify OpenStack to be able to create global instance groups and multi-group instance memberships, as described in Section 5.5.2. These new functions are supported through the Nova Python client and the OpenStack command line client. Hence, the placement logic has been implemented as a patch that modifies the Nova component, the Nova Python client, and the OpenStack command line client.

To use the placement logic, the patch with the ACTiCLOUD code needs to be applied in an OpenStack Pike installation. The new features can be tested through the OpenStack CLI. An example of how to create an instance that belongs to one global group and one normal instance group follows next:

```
% openstack server group create --policy anti-affinity ha-app
% openstack server group create --global --policy soft-anti-affinity gold
% openstack server create ... --hint group=[UUID] --hint group=[UUID] [VM name]
```

To add the VM instance to a third group, the OpenStack CLI can also be used in the following way:

```
% openstack server group create --global --policy soft-anti-affinity noisy
% openstack server group add member [group UUID] [VM name]
```

These APIs are used internally by the ACTiManager in various ways. The ACTiManager uses the add and remove member functionality to modify a VM's group membership after it has been analyzed in the "laboratory" environment to classify the behavior of the VM (e.g. "noisy", "sensitive", etc). In case ACTiManager decides to migrate a VM instance (either from the "laboratory" node to the production nodes or from a production node to a different one), the scheduler will consider the placement policies of all three groups when picking the host destination. Finally, the cloud administrator is also able to create prioritization groups and give non-admin users the option to create VMs with prioritization classes.

## 7 Other Components

This section describes other components that have been developed for both generic and specific hardware platforms (other than those provided by project's partners) which have not been currently integrated with ACTiManager.internal, but are planned to be integrated in the near future.

More specifically, to further support VM prioritization and preserve better performance for gold high-priority VMs that are co-located together with silver best-effort VMs on the same server we describe: (i) a dynamic cache partitioning scheme, (ii) a mechanism for dynamically redistributing huge pages between VMs, and (iii) a low-overhead monitoring mechanism for tracking the active working set of applications. The cache partitioning scheme targets specific hardware platforms that provide support for cache partitioning at the last level of the cache hierarchy. The redistribution mechanism targets generic hardware platforms that provide hardware support for huge pages for better performance. Finally, the low-overhead monitoring mechanism tracks the amount of memory that the applications/VMs use over a specified period. The active working set information could be used to understand the real memory usage of VMs, instead of the amount of initially requested memory, and drive memory overcommitment policies for better co-location of VMs.

### 7.1 Cache Partitioning

Sharing resources on multicore systems without any regulation can be damaging; when multiple applications execute simultaneously, resource contention can lead to destructive interference, unfairness or starvation, and thus reduced and unpredictable performance.

Our goal is to safeguard the QoS of gold VMs while maximising the throughput of silver VMs. We focus on the resource allocation problem where a gold VM is co-located together with multiple silver VMs on the same server. We differentiate from previous works, as our goal is to provide a practical scheme that operates transparently to the running applications, i.e., without any assumption on the information provided by the application itself [LCG15] as well as any profiling information either pre-existing [TMS11, ZE16] or extracted by accompanying profile-inferring tools [MTH11]. This way, our scheme will be able to operate “out-of-the-box” in all typical execution platforms.

To cope with the matter at issue, one can follow a naive, conservative approach and avoid workload consolidation, safeguarding the performance of the gold application; this is clearly a suboptimal approach in terms of effective utilisation of resources. The other extreme approach would be to disregard the priorities of applications and co-locate them within the same server in an unmanaged way, taking the risk of severely harming the gold VM.

A more elaborate approach can leverage the recently introduced support for managing shared hardware resources on modern servers. Intel has released as part of its latest Xeon processors the Intel Resource Director Technology (RDT)<sup>18</sup>, a framework that monitors and manages the shared last-level cache (LLC) and memory bandwidth. Similarly, Cavium has added support for managing

---

<sup>18</sup> Intel Resource Director Technology. <https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html>.

the LLC in the ThunderX processors<sup>19</sup>. Such technologies provide a straightforward mechanism to address our problem: the cache can be taken over almost entirely by the gold VM leaving only a minimum fraction assigned to the silver VMs.

We have experimented with both straightforward cache allocation policies, namely Unmanaged (UM) and Cache Takeover (CT), and made three observations that drive our approach. First, in many multiprogrammed workloads, when allocated exclusively a portion of the cache, the gold VM is able to achieve similar performance to when running alone in the system occupying the entire cache. Hence, there is ample opportunity for dynamically co-locating gold and silver applications, increasing the platform's utilisation. Second, there exist multiprogrammed workloads for which the gold application performs better when allocated exclusively less cache space. While this seems to be counter-intuitive, it happens because the containment of silver applications in minor portion of the cache introduces bandwidth saturation on the memory link that in turn impacts the gold application. Third, strict cache allocation policies that favour the gold application performance result in waste of resources, hurting system utilisation due to their unfairness. On the other hand, leaving resources unmanaged may increase the system's utilisation, at the expense of frequently disrespecting the QoS requirements of the gold application.

We propose DICER [NPG19], a dynamic cache partitioning scheme that targets both system utilisation increase and gold application's performance close to that of isolated execution, i.e., respect the gold application's performance requirements. This is achieved by diligently managing the LLC allocations of the co-located applications, trying to alleviate the effects of contention. We implement DICER based on Intel's RTD support for monitoring and partitioning LLC occupancy and monitoring memory bandwidth.

Our evaluation results on multiprogrammed workloads show that DICER enables the gold application to achieve an SLO of 80% for more than 90% of our workloads and an SLO of 90% for 74% of our workloads; at the same time DICER maintains the effective system utilisation of a full server to 60% on average. Compared to CT, the static cache allocation policy that conservatively favours the gold application, DICER achieves similar or higher conformance to various SLOs, especially as more silver applications are co-located on the same server. At the same time, by dynamically assigning spare cache space to silver applications, DICER achieves in most cases comparable effective utilisation to UM, the unmanaged policy that does not isolate the gold application and enforces no priorities on the consolidated workloads.

## 7.2 Managing and Redistributing Huge pages

To meet the needs of modern applications with large memory footprints, computing platforms have increased their DRAM capacities [FAK12]. However, increased memory capacities introduce a considerable challenge for address translation, i.e., the mappings from the virtual to the physical address space that the operating system/hypervisor manages [BGJ13]. All modern commodity processors use page tables for address translations at software level, and TLBs to cache virtual-to-physical mappings at hardware level. The TLB reach though is difficult to keep up with DRAM trends, since it is very challenging to increase the number of TLB entries without adding extra

---

<sup>19</sup> Cavium ThunderX family of workload optimized processors.  
[https://cavium.com/pdffiles/ThunderX\\_PB\\_p12\\_Rev1.pdf](https://cavium.com/pdffiles/ThunderX_PB_p12_Rev1.pdf)

latency and energy overheads. As a result, large memory workloads experience performance overhead from address translation due to TLB misses when using base 4KB pages. The problem becomes more severe with virtualization that introduces an additional layer of address translation from the guest to the host physical address space.

In response, vendors have equipped modern processors with increased support for larger page sizes or *huge pages* (i.e., 2MB pages). Huge pages increase the TLB reach, reduce the frequency of TLB misses, and hence improve the performance of memory hungry workloads. Despite that increased hardware support, huge pages may provide unsatisfactory performance due to inadequate memory management algorithms on the OS/hypervisor side<sup>20,21,22</sup>. Thus, the burden for improved address translation performance has shifted from the hardware to the system software<sup>23</sup>.

OS-based memory management algorithms have to balance complex trade-offs between address translation overheads, page fault latency, memory bloat and fairness. Ingens [KYP16] is a state-of-the-art OS/hypervisor memory manager that is better than Linux at handling the trade-offs associated with huge page management. In summary: (i) Ingens uses an adaptive policy to balance address translation overhead and memory bloat. (ii) To avoid high page fault latency, Ingens only allocates huge pages in the background with a dedicated kernel thread. (iii) In contrast to Linux, Ingens treats memory contiguity as a resource and employs a share-based policy to allocate huge pages fairly.

We particularly focus on fairness. Fair huge page distribution is important in cloud scenarios, as purchased VMs of the same type are expected to perform equally or the cloud provider's customers have good reason to expect similar performance from identical VM instances. Similarly, VMs of high priority, i.e., gold VMs, are expected to perform better than silver VMs. In such scenarios, the available huge pages have to be shared fairly or adequately proportionally across all the virtual machine instances.

We demonstrate that in memory pressure scenarios, where memory compaction fails to generate enough free space for further huge page allocations, Ingens is unable to achieve a fair state when VMs of the same priority are spawned with a time offset or when their priorities change dynamically.

To cope with those issues, we introduce HPRM, a huge page redistribution mechanism that is triggered automatically when Ingens is “stuck” in an unfair state. When triggered, HPRM reclaims the least frequently used huge pages, so that Ingens can distribute them in a way to reverse the unfair state. HPRM follows Ingens' line, and thus all its functionality takes place in the background. Our evaluation shows that, in contrast to default Ingens, Ingens with HPRM achieves a fair state even when huge pages are limited and processes spawn with a time offset.

---

<sup>20</sup> khugepaged eating 100% CPU. [https://bugzilla.redhat.com/show\\_bug.cgi?id=879801](https://bugzilla.redhat.com/show_bug.cgi?id=879801)

<sup>21</sup> Arch Linux becomes unresponsive from khugepaged. <http://unix.stackexchange.com/questions/161858/arch-linux-becomes-unresponsive-from-khugepaged>.

<sup>22</sup> The Black Magic Of Systematically Reducing Linux OS Jitter. <http://highscalability.com/blog/2015/4/8/the-black-magic-of-systematically-reducing-linux-os-jitter.html>

<sup>23</sup> Transparent Hugepages. <https://lwn.net/Articles/359158/>



### 7.3 Monitoring Active Working Set

In modern computer systems that comprise datacenters, memory is one of the most important and costly components. However, users typically tend to overestimate the memory needs of their VMs/applications. In addition, VMs do not necessarily use as much memory at any one point as they are assigned. These behaviors result in waste of memory or memory underutilization, increase memory pressure, and reduce the number of VMs that can be co-located on the same server<sup>24</sup>.

Memory overcommitment allows the assignment of more memory to VMs than the available physical memory of the server. For example, if a VM requests for 4 GB of memory on a physical machine, but it is only using 500 MB, then it is possible to create additional VM(s) that take advantage of the free 3.5 GB. However, to drive memory overcommitment policies for better co-location of VMs, an efficient and low-overhead mechanism for measuring precisely the actively used memory of each of the VMs, or otherwise their working set<sup>25</sup> [PD68], is needed.

We study in detail some of the techniques suggested in the literature for measuring the active working set of VMs/applications [NKY18]. First, we focused on the idle page tracking technique<sup>26</sup> that relies on the functionality and the API added to Linux since kernel version 4.3. Idle page tracking allows a resource manager to detect which and how many memory pages have been recently accessed by the VM or have remained idle. In addition, we implemented two previously proposed measurement techniques, one based on sampling [CW02] and another based on hardware performance counters [ZJW11], that aim at reducing the performance overhead. We then focused on extending the Linux kernel by installing a new interface for taking measurements of the working set of applications using both idle page tracking and sampling for better performance. We evaluated the proposed mechanisms and techniques on a Linux server with QEMU/KVM as a hypervisor. Our experimental results show that the proposed techniques minimize the performance overhead of tracking the active working sets of VMs without loss in the accuracy of measurements.

---

<sup>24</sup> <https://lwn.net/Articles/787611/>

<sup>25</sup> [https://en.wikipedia.org/wiki/Working\\_set](https://en.wikipedia.org/wiki/Working_set)

<sup>26</sup> [https://www.kernel.org/doc/html/latest/admin-guide/mm/idle\\_page\\_tracking.html](https://www.kernel.org/doc/html/latest/admin-guide/mm/idle_page_tracking.html)

## 8 Summary

This document describes the final version 2.0 of ACTiManager and the different components implemented under the ACTiManager. The document also presents the detailed design decisions for each component in ACTiManager as well as modifications and integration points with OpenStack and other ACTiCLOUD components. Future work includes the integration of ACTiManager with MicroVisor, MonetDB and end-to-end evaluation of the system.

The source code of ACTiManager is publicly available at the following repository:

<https://github.com/acticloud/actimanager>



## 9 References

- [ACTi\_D1.1] D1.1: “ACTiCLOUD Requirements and base architecture”, ACTiCLOUD deliverable.
- [ACTi\_D1.2] D1.2: “ACTiCLOUD architecture”, ACTiCLOUD deliverable.
- [ACTi\_D2.2] D2.2: “Distributed Cloud Resource Manager v1.0”, ACTiCLOUD deliverable.
- [AKP18] Fanourios Arapidis, Vasileios Karakostas, Nikela Papadopoulou, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. "Performance Prediction of NUMA Placement: A Machine-Learning Approach". In 1st International Workshop on Next Generation Clouds for Extreme Data (XtremeCLOUD 2018) - held in conjunction with CloudCom 2018. doi: 10.1109/CloudCom2018.2018.00064
- [APG18] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. 2018. On the diversity of cluster workloads and its impact on research results. In 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston, MA, 533–546. <https://www.usenix.org/conference/atc18/presentation/amvrosiadis>
- [BGC13] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient virtual memory for big memory servers. In Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13). ACM, New York, NY, USA, 237–248. DOI: <http://dx.doi.org/10.1145/2485922.2485943>
- [BF11] Sergey Blagodurov and Alexandra Fedorova. 2011. In Search for Contention-descriptive Metrics in HPC Cluster Environment. In Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering (ICPE '11). ACM, New York, NY, USA, 457–462. <https://doi.org/10.1145/1958746.1958815>
- [BSA17] A. Bhattacharyya, S. Sotiriadis, and C. Amza. 2017. Online Phase Detection and Characterization of Cloud Applications. In 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). 98–105. <https://doi.org/10.1109/CloudCom.2017.21>
- [BZF10] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. 2011. A case for NUMA-aware contention management on multicore systems. In Proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIXATC'11). USENIX Association, Berkeley, CA, USA, 1–1.
- [BTS13] Alex D. Breslow, Ananta Tiwari, Martin Schulz, Laura Carrington, Lingjia Tang, and Jason Mars. 2013. Enabling fair pricing on HPC systems with node sharing. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13). ACM, New York, NY, USA, Article 37, 12 pages. DOI: <https://doi.org/10.1145/2503210.2503256>
- [CAK16] Ignacio Cano, Srinivas Aiyar, and Arvind Krishnamurthy. 2016. Characterizing Private Clouds: A Large-Scale Empirical Analysis of Enterprise Clusters. In Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16). ACM, New York, NY, USA, 29–41. <https://doi.org/10.1145/2987550.2987584>
- [CSS14] Lihua Chen, Haiying Shen, and Karan Sapra. 2014. Distributed Autonomous Virtual Resource Management in Datacenters Using Finite-Markov Decision Process. In Proceedings of the

ACM Symposium on Cloud Computing (SOCC '14). ACM, New York, NY, USA, Article 24, 13 pages. <https://doi.org/10.1145/2670979.2671003>

[CWL19] Quan Chen, Zhenning Wang, Jingwen Leng, Chao Li, Wenli Zheng, and Minyi Guo. 2019. Avalon: towards QoS awareness and improved utilization through multi-resource management in datacenters. In Proceedings of the ACM International Conference on Supercomputing (ICS '19). ACM, New York, NY, USA, 272–283. DOI: <https://doi.org/10.1145/3330345.3330370>

[CBM17] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17). ACM, New York, NY, USA, 153–167. <https://doi.org/10.1145/3132747.3132772>

[CLL15] Chen Chen, Changbin Liu, Pingkai Liu, Boon Thau Loo, and Ling Ding. 2015. A scalable multi-datacenter layer-2 network architecture. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15). ACM, New York, NY, USA, Article 8, 12 pages. DOI: <https://doi.org/10.1145/2774993.2775008>

[CS00] Nello Cristianini, John Shawe-Taylor, et al. 2000. An introduction to support vector machines and other kernel-based learning methods. Cambridge university press.

[CW02] Carl A. Waldspurger. “Memory Resource Management in VMware ESX Server”. In: SIGOPS Oper. Syst. Rev. 36.SI (Dec. 2002), pp. 181–194. issn: 0163-5980. doi: 10.1145/844128.844146. url: <http://doi.acm.org/10.1145/844128.844146>

[DNG14] Daniel J. Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. 2014. PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures. In Proceedings of the ACM Symposium on Cloud Computing (SOCC '14). ACM, New York, NY, USA, Article 8, 13 pages. <https://doi.org/10.1145/2670979.2670987>

[DDD18] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2018. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates. In Proceedings of the ACM Symposium on Cloud Computing (SoCC '18). ACM, New York, NY, USA, 135–148. <https://doi.org/10.1145/3267809.3267838>

[DK13] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13). ACM, New York, NY, USA, 77–88. <https://doi.org/10.1145/2451116.2451125>

[DK14] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14). ACM, New York, NY, USA, 127–144. <https://doi.org/10.1145/2541940.2541941>

[DK17] Christina Delimitrou and Christos Kozyrakis. 2017. Bolt: I Know What You Did Last Summer... In The Cloud. In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17). ACM, New York, NY, USA, 599–613. <https://doi.org/10.1145/3037697.3037703>

- [DSK15] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15). ACM, New York, NY, USA, 97–110. <https://doi.org/10.1145/2806777.2806779>
- [DWD13] Tanima Dey, Wei Wang, Jack W. Davidson, and Mary Lou Soffa. 2013. ReSense: Mapping Dynamic Workloads of Colocated Multithreaded Applications Using Resource Sensitivity. ACM Trans. Archit. Code Optim. 10, 4, Article 41 (Dec. 2013), 25 pages. <https://doi.org/10.1145/2555289.2555298>
- [DFB12] Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, Mark Roth, Fabien Gaud, and Jian Pei. 2012. A practical method for estimating performance degradation on multicore processors, and its application to HPC workloads. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12). IEEE Computer Society Press, Los Alamitos, CA, USA, , Article 83 , 11 pages.
- [FAK12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafae, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII). ACM, New York, NY, USA, 37–48. DOI=<http://dx.doi.org/10.1145/2150976.2150982>
- [GZH19] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19). ACM, New York, NY, USA, 129–142. <https://doi.org/10.1145/3297858.3304004>
- [GKP18] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In Proceedings of the Thirteenth EuroSys Conference (EuroSys '18). ACM, New York, NY, USA, Article 4, 13 pages. <https://doi.org/10.1145/3190508.3190549>
- [GVH17] Daniel Goodman, Georgios Varisteas, and Tim Harris. 2017. Pandia: Comprehensive Contention-sensitive Thread Placement. In Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17). ACM, New York, NY, USA, 254–269. <https://doi.org/10.1145/3064176.3064177>
- [GLK11] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. 2011. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines. In Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11). ACM, New York, NY, USA, Article 22, 14 pages. <https://doi.org/10.1145/2038916.2038938>
- [HGA14] A. Haritatos, G. Goumas, N. Anastopoulos, K. Nikas, K. Kourtis, and N. Koziris. 2014. LCA: A memory link and cache-aware co-scheduling approach for CMPs. In 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT). 469–470. <https://doi.org/10.1145/2628071.2628123>

- [HKG16] Alexandros-Herodotos Haritatos, Konstantinos Nikas, Georgios I. Goumas, and Nectarios Koziris. 2016. A resource-centric Application Classification Approach. In COSH@HiPEAC. TUM Library, 7–12. <http://doi.org/10.14459/2016md1286948>
- [HGY15] Chien-Chun Hung, Leana Golubchik, and Minlan Yu. 2015. Scheduling Jobs Across Geo-distributed Datacenters. In Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15). ACM, New York, NY, USA, 111–124. <https://doi.org/10.1145/2806777.2806780>
- [IAK18] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-sensitive Services. In Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18). USENIX Association, Berkeley, CA, USA, 519–531. <http://dl.acm.org/citation.cfm?id=3277355.3277406>
- [JS17] Preeti N Jain and Sunil K Surve, "Modeling resource constrained solo applications using logistic growth model," 2017 International Conference on Advances in Computing, Communication and Control (ICAC3), Mumbai, 2017, pp. 1-8. doi: 10.1109/ICAC3.2017.8318751
- [KBH08] R. Knauerhase, P. Brett, B. Hohlt, T. Li and S. Hahn, "Using OS Observations to Improve Performance in Multicore Systems," in IEEE Micro, vol. 28, no. 3, pp. 54-66, May-June 2008. doi: 10.1109/MM.2008.48
- [KGL18] Vasileios Karakostas, Georgios Goumas, Ewnetu Bayuh Lakew, Erik Elmroth, Stefanos Gerangelos, Simon Kolberg, Konstantinos Nikas, Stratos Psomadakis, Dimitrios Siakavaras, Petter Svärd, and Nectarios Koziris. 2018. Efficient resource management for data centers: the ACTiCLOUD approach. In Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS '18). ACM, New York, NY, USA, 244-246. DOI: <https://doi.org/10.1145/3229631.3236095>
- [KMH12] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. 2012. Measuring Interference Between Live Datacenter Applications. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12). IEEE Computer Society Press, Los Alamitos, CA, USA, Article 51, 12 pages. <http://dl.acm.org/citation.cfm?id=2388996.2389066>
- [KJL18] R. S. Kannan, A. Jain, M. A. Laurenzano, L. Tang, and J. Mars. 2018. Proctor: Detecting and Investigating Interference in Shared Datacenters. In 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). 76–86. <https://doi.org/10.1109/ISPASS.2018.00016>
- [KLA19] Ram Srivatsa Kannan, Michael Laurenzano, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. Caliper: Interference Estimator for Multi-tenant Environments Sharing Architectural Resources. ACM Trans. Archit. Code Optim. 16, 3, Article 22 (June 2019), 25 pages. DOI: <https://doi.org/10.1145/3323090>
- [KGB13] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. 2013. ADAPT: A framework for co-scheduling multithreaded programs. ACM Trans. Archit. Code Optim. 9, 4, Article 45 (Jan. 2013), 24 pages. <https://doi.org/10.1145/2400682.2400704>

- [KYP16] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16). USENIX Association, Berkeley, CA, USA, 705–721. <http://dl.acm.org/citation.cfm?id=3026877.3026931>
- [LWZ10] Young Choon Lee, Chen Wang, Albert Y. Zomaya, and Bing Bing Zhou. 2010. Profit-Driven Service Request Scheduling in Clouds. In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID '10). IEEE Computer Society, Washington, DC, USA, 15–24. DOI: <https://doi.org/10.1109/CCGRID.2010.83>
- [LLD08] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," 2008 IEEE 14th International Symposium on High Performance Computer Architecture, Salt Lake City, UT, 2008, pp. 367–378. doi: 10.1109/HPCA.2008.4658653
- [LCG14] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. In Proceeding of the 41st annual international symposium on Computer architecture (ISCA '14). IEEE Press, Piscataway, NJ, USA, 301–312.
- [LCG15] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, 2015, pp. 450–462. doi: 10.1145/2749469.2749475
- [LPM17] Ewnetu Bayuh Lakew, Alessandro Vittorio Papadopoulos, Martina Maggio, Cristian Klein, and Erik Elmroth. 2017. KPI-agnostic Control for Fine-Grained Vertical Elasticity. In Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '17). IEEE Press, Piscataway, NJ, USA, 589–598. DOI: <https://doi.org/10.1109/CCGRID.2017.71>
- [LLD08] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," 2008 IEEE 14th International Symposium on High Performance Computer Architecture, Salt Lake City, UT, 2008, pp. 367–378. doi: 10.1109/HPCA.2008.4658653
- [MT13] Jason Mars and Lingjia Tang. 2013. Whare-map: Heterogeneity in "Homogeneous" Warehouse-scale Computers. In Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13). ACM, New York, NY, USA, 619–630. <https://doi.org/10.1145/2485922.2485975>
- [MTH11] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44). ACM, New York, NY, USA, 248–259. <https://doi.org/10.1145/2155620.2155650>
- [MSB10] Andreas Merkel, Jan Stoess, and Frank Bellosa. 2010. Resource-conscious scheduling for energy efficiency on multicore processors. In Proceedings of the 5th European conference on Computer systems (EuroSys '10). ACM, New York, NY, USA, 153–166. DOI=<http://dx.doi.org/10.1145/1755913.1755930>

- [NIG07] Ripal Nathuji, Canturk Isci, and Eugene Gorbato. 2007. Exploiting Platform Heterogeneity for Power Efficient Data Centers. In Proceedings of the Fourth International Conference on Autonomic Computing (ICAC '07). IEEE Computer Society, Washington, DC, USA, 5–. <https://doi.org/10.1109/ICAC.2007.16>
- [NKG10] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds. In Proceedings of the 5th European Conference on Computer Systems (EuroSys '10). ACM, New York, NY, USA, 237–250. <https://doi.org/10.1145/1755913.1755938>
- [NKY18] Vlad Nitu et al. “Working Set Size Estimation Techniques in Virtualized Environments: One Size Does Not Fit All”. In: Proc. ACM Meas. Anal. Comput. Syst. 2.1 (Apr. 2018), 19:1–19:22. issn: 2476-1249. doi: 10.1145/3179422. url: <http://doi.acm.org/10.1145/3179422>.
- [NPG19] Konstantinos Nikas, Nikela Papadopoulou, Dimitra Giantsidi, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. 2019. DICER: Diligent Cache Partitioning for Efficient Workload Consolidation. In Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019). ACM, New York, NY, USA, Article 15, 10 pages. DOI: <https://doi.org/10.1145/3337821.3337891>
- [NCP17] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell. 2017. Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). 409–420. <https://doi.org/10.1109/HPCA.2017.13>
- [NVN13] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13). USENIX Association, Berkeley, CA, USA, 219–230. <http://dl.acm.org/citation.cfm?id=2535461.2535489>
- [PD68] Peter J. Denning. “The Working Set Model for Program Behavior”. In: Commun. ACM 11.5 (May 1968), pp. 323–333. issn: 0001-0782. doi: 10.1145/363095.363141. url: <http://doi.acm.org/10.1145/363095.363141>.
- [PGS19] Stratos Psomadakis, Stefanos Gerangelos, Dimitrios Siakavaras, Ioannis Papadakis, Marina Vemmou, Aspa Skalidi, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Georgios Goumas. "ACTiManager: An end-to-end interference-aware cloud resource manager". In 20th International Middleware Conference Demos and Posters (Middleware 2019).
- [PNK17] Ioannis Papadakis, Konstantinos Nikas, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. 2017. Improving QoS and Utilisation in modern multi-core servers with Dynamic Cache Partitioning. In Proceedings of the Joined Workshops COSH 2017 and VisorHPC 2017. <http://doi.org/10.14459/2017md1344298>
- [KP01] Karl Pearson. 1901. LIII. On lines and planes of closest fit to systems of points in space. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 2, 11 (1901), 559–572.

- [RBG12] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. 2012. Generalized Resource Allocation for the Cloud. In Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12). ACM, New York, NY, USA, Article 15, 12 pages. <https://doi.org/10.1145/2391229.2391244>
- [RKC16] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. 2016. PerfOrator: Eloquent Performance Models for Resource Optimization. In Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16). ACM, New York, NY, USA, 415–427. <https://doi.org/10.1145/2987550.2987566>
- [RTG12] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12). ACM, New York, NY, USA, Article 7, 13 pages. <https://doi.org/10.1145/2391229.2391236>
- [RTM10] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silviu Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. IEEE Micro 30, 4 (July 2010), 65–79. <https://doi.org/10.1109/MM.2010.68>
- [RD18] Francisco Romero and Christina Delimitrou. 2018. Mage: Online and Interference-aware Scheduling for Multi-scale Heterogeneous Systems. In Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18). ACM, New York, NY, USA, Article 19, 13 pages. <https://doi.org/10.1145/3243176.3243183>
- [R87] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. Journal of computational and applied mathematics 20 (1987), 53–65.
- [RDK11] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing (CLOUD '11). IEEE Computer Society, Washington, DC, USA, 500–507. <https://doi.org/10.1109/CLOUD.2011.42>
- [RKG13] Alan Roytman, Aman Kansal, Sriram Govindan, Jie Liu, and Suman Nath. 2013. PACMan: Performance aware virtual machine consolidation. In Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13). 83–94.
- [SSG11] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11). ACM, New York, NY, USA, Article 5, 14 pages. <https://doi.org/10.1145/2038916.2038921>
- [SLN12] ChunYi Su, Dong Li, Dimitrios Nikolopoulos, Matthew Grove, Kirk W. Cameron, and Bronis R. de Supinski. 2011. Critical path-based thread placement for NUMA systems. In Proceedings of the second international workshop on Performance modeling, benchmarking and simulation of high performance computing systems (PMBS '11). ACM, New York, NY, USA, 19–20. DOI: <https://doi.org/10.1145/2088457.2088471>
- [SH11] Weiming Shi and Bo Hong. 2011. Towards Profitable Virtual Machine Placement in the Data Center. In Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing (UCC '11). IEEE Computer Society, Washington, DC, USA, 138–145.

- [SFS12] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12). USENIX Association, Berkeley, CA, USA, 349–362. <http://dl.acm.org/citation.cfm?id=2387880.2387914>
- [SSG15] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory. In Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48). ACM, New York, NY, USA, 62–75. <https://doi.org/10.1145/2830772.2830803>
- [TMS11] Lingjia Tang, Jason Mars, and Mary Lou Soffa. 2011. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT '11). ACM, New York, NY, USA, 12–21. <https://doi.org/10.1145/2000417.2000419>
- [TGS14] Priyanka Tembey, Ada Gavrilovska, and Karsten Schwan. 2014. Merlin: Application- and Platform-aware Resource Allocation in Consolidated Server Systems. In Proceedings of the ACM Symposium on Cloud Computing (SOCC '14). ACM, New York, NY, USA, Article 14, 14 pages. <https://doi.org/10.1145/2670979.2670993>
- [VPK15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15). ACM, New York, NY, USA, Article 18, 17 pages. <https://doi.org/10.1145/2741948.2741964>
- [WVC11] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, and K. Schwan. 2011. Statistical techniques for online anomaly detection in data centers. In 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops. 385–392. <https://doi.org/10.1109/INM.2011.5990537>
- [WKB11] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. 2011. SLA-Based Resource Allocation for Software as a Service Provider (SaaS) in Cloud Computing Environments. In Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '11). IEEE Computer Society, Washington, DC, USA, 195–204. DOI: <https://doi.org/10.1109/CCGrid.2011.51>
- [XL08] Y. Xie and G. H. Loh. 2008. Dynamic Classification of Program Memory Behaviors in CMPs. In Proceedings of Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI).
- [YAK14] Neeraja J. Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. 2014. Wrangler: Predictable and Faster Jobs Using Fewer Resources. In Proceedings of the ACM Symposium on Cloud Computing (SOCC '14). ACM, New York, NY, USA, Article 26, 14 pages. <https://doi.org/10.1145/2670979.2671005>
- [YBM13] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubbleflux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13). ACM, New York, NY, USA, 607–618. <https://doi.org/10.1145/2485922.2485974>



- [ZCX15] Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. 2016. Predicting Cross-Core Performance Interference on Multicore Processors with Regression Analysis. *IEEE Trans. Parallel Distrib. Syst.* 27, 5 (May 2016), 1443-1456. DOI=<http://dx.doi.org/10.1109/TPDS.2015.2442983>
- [ZDL17] Jinshi Zhang, Eddie Dong, Jian Li, and Haibing Guan. 2017. MigVisor: Accurate Prediction of VM Live Migration Behavior using a Working-Set Pattern Model. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*. ACM, New York, NY, USA, 30-43. DOI: <https://doi.org/10.1145/3050748.3050753>
- [ZJW11] Weiming Zhao et al. “Low Cost Working Set Size Tracking”. In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference. USENIXATC'11*. Portland, OR: USENIX Association, 2011, pp. 17–17. url: <http://dl.acm.org/citation.cfm?id=2002181.2002198>.
- [ZBF10] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 129–142. <https://doi.org/10.1145/1736020.1736036>
- [ZE16] Haishan Zhu and Mattan Erez. 2016. Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 33-47. DOI: <https://doi.org/10.1145/2872362.2872394>