



ACTiCLOUD: ACTivating resource efficiency and large databases in the CLOUD

Project No: 732366

H2020-ICT-2016-1

D4.5: ACTiCLOUD Final Evaluation

Due date of deliverable:	M36 (2019/12/31)
Actual submission date:	M37 (2020/01/31)

Executive summary:

Deliverable D4.5: "ACTiCLOUD Final Evaluation" describes the evaluation of the final ACTiCLOUD prototype and its constituent components that were described in Deliverable D4.4 "ACTiCLOUD Final Prototype". The evaluation is based on the strategy and methodology that was described in D4.3 "ACTiCLOUD intermediate evaluation", regarding the strategic objectives of the project and the use cases and business scenarios that the ACTiCLOUD architecture targets.

List of authors:

Author	Affiliation
Jim Webber	NEO
Georgios Goumas, Vasileios Karakostas, Dimitrios Siakavaras, Stefanos Gerangelos, Stratos Psomadakis	ICCS
Atle Vesterkjær, Daniel J Blueman	NSCALE
Michail Flouris, Stelios Louloudakis	ONAPP
Christos Kotselidis and Foivos Zakkak	UNIMAN
Ewnetu Bayuh Lakew	UMU
Joeri van Ruth, Panagiotis Koutsourakis, Ying Zhang, Martin Kersten	MDBS
Monica Vatteroni	KALEAO

Dissemination Level	X	PU (Public)
		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:
Nature	X	R (Report)
		P (Prototype)
		D (Demonstrator)
		O (Other)

Review Status		Draft
		WP Leader accepted
		QA approved
	X	Coordinator accepted

Revision History:

Version	Author(s) (Affiliation)	Notes
0.1	Jim Webber (NEO)	Initial ToC
0.2	All authors	Content updated
0.3	Foivos Zakkak (UNIMAN), Ewnetu Bayuh Lakew (UMU), Jim Webber (NEO)	Document reviewed
0.4	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

KALEAO

onapp



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

1	About ACTiCLOUD	10
1.1	Purpose of this Document	10
1.2	Document Structure	10
2	Evaluation Strategy.....	11
2.1	Resource Efficiency	11
2.2	Performance Stability	12
2.3	Scalability in Resource Provisioning	13
2.4	Elasticity in Resource Provisioning	13
3	Resource efficiency	15
3.1	ACTiManager	15
3.1.1	Methodology	15
3.1.2	Results	19
3.1.3	Summary	22
3.2	HyperVisor	22
3.2.1	Benchmarks and Metrics	23
3.2.2	Evaluation of the MicroVisor on large Intel x86 servers	25
3.2.3	Evaluation of the MicroVisor on the KMAX platform.....	29
3.3	Accelerated storage access on KMAX	34
3.4	Resource efficiency with KMAX	34
4	Performance stability.....	35
4.1	ACTiManager Generic Internal and External Components.....	35
4.1.1	Methodology	35
4.1.2	Results	35
4.1.3	Summary	37
4.2	ACTiManager.Internal on Numascale.....	37
4.3	ACTiManager.external.....	40
5	Scalability in Resource Provisioning	42
5.1	System Libraries	42
5.1.1	NC-ALLOC.....	42
5.1.2	OpenMP Application.....	43
5.1.3	KVM.....	45
5.1.4	NumaConnect™ Byte Transfer Layer (NC-BTL)	50
5.1.5	NC-LAPACK	54
5.1.6	Conclusion	55
5.2	Neo4j on the ACTiCLOUD platform	55
5.2.1	Neo4j on Numascale system with multiple servers.....	55
5.2.2	Neo4j on Numascale system with a single server	57
5.3	MonetDB on the ACTiCLOUD platform	58
5.3.1	From 2017 to 2019 and beyond	59
5.3.2	MonetDB in the cloud.....	60
5.3.3	Scale-down on KMAX	64
5.3.4	Summary.....	65
5.4	Hyperscale JVM	65
5.4.1	Evaluation platforms	66
5.4.2	Evaluation methodology	66
5.4.3	Performance evaluation against HotSpot, the standard production-quality JVM.....	66
5.4.4	Performance evaluation against JikesRVM, the competing research VM.....	69
5.4.5	Memory Profiler Evaluation	71
5.4.6	Conclusions.....	75
6	Elasticity in Resource Provisioning	76

7 Conclusions 79

Figures

Figure 3.1: Pricing models for gold and silver applications.	17
Figure 3.2: Datacenter throughput (a) and application slowdown (b). The top figure (a) shows the throughput, i.e., the total number of all accepted VMs' vCPUs, broken down into gold and silver vCPUs. Numbers within the bars designate the number of gold/silver VMs that respected their SLAs. The number in the rectangle shows the percentage of the compute resources utilized (shown only for ACTiManager as other policies use 100% of resources). The bottom figure (b) shows the average slowdown of applications.....	20
Figure 3.3: Profit per resource management scheme. The numbers inside the bars designate the number of gold and silver VMs that respected their SLAs. The numbers in the rectangles show the percentage of the compute resources utilized.	22
Figure 3.4: The results of inter-VM network throughput for MicroVisor (MV) and KVM.	25
Figure 3.5: The results of inter-VM network latency for MicroVisor (MV) and KVM.	26
Figure 3.6: UnixBench system performance measurements.....	27
Figure 3.7: STREAM memory bandwidth measurements.....	29
Figure 3.8: KMAX results for SysBench CPU benchmark.	30
Figure 3.9: KMAX Results for SysBench Memory benchmark.....	31
Figure 3.10: KMAX results for Sysbench OLTP Select point benchmark.	31
Figure 3.11: KMAX results for SysBench OLTP Random_Range benchmark.....	32
Figure 3.12: KMAX results for SysBench OLTP Write_Only Benchmark.....	32
Figure 3.13: KMAX results for SysBench OLTP Read_Only benchmark.	33
Figure 3.14: KMAX results for SysBench OLTP Read_Write benchmark.....	33
Figure 3.15: Impact on CEPH distributed storage from Fast Path Integration of SSD.	34
Figure 4.1: (Combines Figures 3.2a and 3.2b) Datacenter throughput and application slowdown. The primary axis (left) shows the total number of all accepted VMs' vCPUs, broken down into gold and silver vCPUs. The secondary axis (right) shows the average slowdown of applications. Numbers within the bars designate the number of gold/silver VMs that respected their SLAs. The number in the rectangle shows the percentage of the compute resources utilized (shown only for ACTiManager as other policies use 100% of resource).....	35
Figure 4.2: Relative performance (IPC, MPI, and application performance) for different applications using vanilla algorithm.....	39
Figure 4.3: Relative performance (IPC, MPI, and application performance) for different applications using SM-MPI.....	39
Figure 4.4: Relative performance (IPC, MPI, and application performance) for different applications using SM-IPC.....	40
Figure 4.5: In-site load-balancing and cross-site offloading.	41
Figure 5.1: NPB OMP EP.....	44
Figure 5.2: Stream benchmark on the Numascale Shared Memory Server in Athens.	45
Figure 5.3: Showing that KVM keeps the Numa topology by running two windows running in parallel (OpenMP NPB EP and htop) by showing that every core is used when this is fed to OpenMP through affinity setting.	47

Figure 5.4: Showing that KVM keeps the Numa topology by running two windows running in parallel (OpenMP NPB EP and htop) by showing that every second core is used when this is fed to OpenMP through affinity settings.....	49
Figure 5.5: Showing that the KVM overhead is negligible on Numascale Shared Memory Installations in ACTiCLOUD.....	50
Figure 5.6: GROMACS Scaling on the Numascale Shared Memory system in ACTiCLOUD.	51
Figure 5.7: NAS Parallel Benchmark SP.....	53
Figure 5.8: NAS Parallel Benchmark BT.	53
Figure 5.9: NAS Parallel Benchmark LU.	54
Figure 5.10: LDBC SNB Q3 throughput with cache size 150G and 600G.....	56
Figure 5.11: NumaScope screenshot.....	57
Figure 5.12: TPC-H query execution times in seconds with 1TB data (i.e. SF1000) and MonetDB Apr2019-SP1 release: MicroVisor VM versus AWS EC2 VM versus NSCALE bare-metal.....	62
Figure 5.13: SF250 on i3.4xlarge and SF1000 on i3.16xlarge: the execution time and price of each query. The “increase” columns show respectively the increases in time and price from SF250 to SF1000 computed by SF1000/SF250 of the corresponding columns.	63
Figure 5.14: TPC-H scale factors 1.0, 0.3, 0.1, 0.03, 0.01, and 0.0 (i.e. 1GB, 300MB, 100MB, 30 MB, 10MB and 0MB) with MonetDB, MonetDBLite, SQLite, MariaDB, PostgreSQL on one KMAX server with 4 ARM cores and 4GB RAM. The numbers shown are queries-per-sec for each query, hence, larger is better.	64
Figure 5.15: Hyperscale JVM vs HotSpot VM using DaCapo-9.12-MR1-bach on Oslo testbed.	67
Figure 5.16: Hyperscale JVM slowdown over Hotspot VM using DaCapo-9.12-MR1-bach on Oslo testbed.	67
Figure 5.17: Hyperscale JVM vs HotSpot VM using LDBC SNB with Neo4j on Oslo testbed.	68
Figure 5.18: Hyperscale JVM slowdown over Hotspot VM using LDBC SNBwith Neo4j on Oslo testbed.	68
Figure 5.19: Hyperscale JVM vs HotSpot VM using DaCapo-9.12-MR1-bach on Manchester testbed.	69
Figure 5.20: Hyperscale JVM slowdown over Hotspot VM using DaCapo-9.12-MR1-bach on Manchester testbed.....	69
Figure 5.21: Hyperscale JVM vs JikesRVM using DaCapo-9.12-bach on Oslo testbed.....	70
Figure 5.22: Hyperscale JVM speedup over JikesRVM using DaCapo-9.12-bach on Oslo testbed (fop, tradebeans and tradesoap fail on JikesRVM).....	71
Figure 5.23: Number of object allocations per benchmark, as measured by AntTracks VM and Hyperscale JVM.	72
Figure 5.24: Comparison between Hypescale JVM without profiling support (No profiling), with profiling support but no active profiling (Not Active), and with profiling support and active profiling (Active).	73
Figure 5.25: Overhead of Hypescale JVM profiling support when not actively profiling.....	74
Figure 5.26: Overhead of Hypescale JVM profiling support when actively profiling.....	74
Figure 6.1: Resizing operation with ACTiManager.	77
Figure 6.2: Resizing operation with default Linux scheduling.....	77
Figure 6.3: Resizing operation with static pinning on big cores.....	78
Figure 6.4: Resizing operation with static pinning on little cores.....	78

Tables

Table 3.1: The SPEC 2006 benchmarks used in our experimental evaluation and their interference classification.....	18
Table 3.2: Pricing policy parameters. In all experiments we use $t_g = 1.2$ and $t_s = 4$	18
Table 4.1: Total number of gold/silver VMs that respected their SLAs, out of the total number of gold/silver VMs that were accepted for execution. For example, 14 gold VMs out of a total of 15 gold VMs respected their SLA (up to 20% slowdown compared to standalone execution) with OpenStack for the “low_20_50” execution scenario.....	36
Table 4.2: VM types used for the experiment.....	37
Table 5.1: allocation/deallocation performance test, al.	43
Table 5.2: NC-ALLOC push/pop performance test, pc.	43
Table 5.3: Eigenvalue solver (N=14000) with NC-LAPACK or MKL.	54
Table 5.4: evolution of MonetDB performance during ACTiCLOUD using TPC-H scale factor 1, 10, and 100 on an Intel desktop. The numbers are the total execution time of all TPC-H 22 queries in seconds, hence, smaller is better. “Default” is the current development branch to be released in H1/2020.	59
Table 5.5: evolution of MonetDBLite performance during ACTiCLOUD using TPC-H scale factor 0, 0.01, 0.03, 0.1, 0.3 and 1 on a KMAX server. The numbers are the total execution time of all TPC-H 22 queries in seconds, hence, smaller is better. “SF1*” is the total ex execution time of SF1 without the problem query 20, which is under investigation.	60
Table 5.6: TPC-H Power@Size performance metric (higher is better).	62

List of Abbreviations

Abbreviation / Acronym	Meaning
CSP	Cloud Service Provider
HV	Hypervisor
IPC	Instructions per Cycle
ISA	Instruction Set Architecture
JVMCI	Java Virtual Machine compiler interface
KPI	Key Performance Indicator
LLC	Last-level cache
NPU	Network Processing Unit
NUMA	Non-uniform memory access
OLAP	OnLine Analytical Processing
OLTP	OnLine Transaction Processing
OS	Operating System
QoS	Quality of Service
SF	Scale factor
SLA	Service Level Agreement
SNB	Social Network Benchmark
SO	Strategic Objective
SoC	System on Chip
SPU	StorageProcessing Unit
TCO	Total Cost of Ownership
VM	Virtual Machine
VMM	Virtual Machine Monitor

1 About ACTiCLOUD

ACTiCLOUD's vision is to develop a novel cloud architecture that will break the existing scale-up and share-nothing barriers and enable the holistic management of physical resources both at the local cloud site and the distributed levels, targeting drastically improved utilization and scalability of resources. This ultimately translates 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 enables the creation of distributed, hyper-converged, “share-anything”, 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

The purpose of this deliverable D4.5 “ACTiCLOUD Final Evaluation” is to evaluate the ACTiCLOUD final prototype with respect to the project's four strategic objectives:

- SO1: Effective utilization of cloud resources, through:
 - SO1.1: Resource efficiency, and
 - SO1.2: Performance stability.
- SO2: Deployment of resource demanding applications in the cloud, through:
 - SO2.1: Scalability in resource provisioning, and
 - SO2.2: Elasticity in resource provisioning.

Finally, in this deliverable we evaluate the final versions v2.0 of the individual components that were integrated together for the ACTiCLOUD final prototype.

1.2 Document Structure

This document is structured as follows. Section 2 describes the evaluation strategy for the ACTiCLOUD final prototype along the four strategic objectives of ACTiCLOUD. Sections 3, 4, 5, and 6 evaluate the resource efficiency, performance stability, scalability in resource provisioning, and elasticity in resource provisioning, respectively. Finally, Section 7 concludes this document and summarizes challenges and future work.

2 Evaluation Strategy

In this section we describe the evaluation strategy of the ACTiCLOUD architecture at high-level. The evaluation strategy is refined based on the deliverable D4.3 "ACTiCLOUD Intermediate Prototype". Our evaluation approach is based on the strategic objectives of the project. The remainder of this section describes for each strategic objective: (i) the Key Performance Indicators (KPIs), (ii) the target values for those KPIs, (iii) the baseline that is used for comparison to show the benefits of the ACTiCLOUD architecture, (iv) the benchmarks, (v) the testbeds, and (vi) the correlation with the business scenarios and application use-cases as they were defined in deliverable D1.1 "ACTiCLOUD requirements and base architecture" and deliverable D1.2 "ACTiCLOUD Architecture".

2.1 Resource Efficiency

Current cloud installations suffer from low utilization levels that leave precious resources unused or underutilized. This is typically done to provide high Quality of Service (QoS). ACTiCLOUD aims to significantly increase the resource efficiency of cloud infrastructures while maintaining the existing Quality of Service (QoS) standards.

KPI: Increasing resource efficiency may translate to various metrics, such as increasing total system throughput per resource item, where resource item can be a hardware item (e.g., #servers, #cores, size of memory, etc), power/energy metric (W, Wh), or cost metric (€).

To evaluate ACTiCLOUD architecture's resource efficiency we employ a metric that describes the gain for the cloud service providers in terms of profit increase thanks to higher utilization of the system (i.e., the increase of the total system throughput), while respecting the Quality of Service (QoS) of the applications/users. The "gain" metric is based on the following assumptions:

- **Billing depends on the prioritization of VMs:** the Cloud Service Providers (CSPs) may distinguish the running Virtual Machines (VMs) between high-priority, latency-critical VMs (i.e., "gold" instances) and low-priority, batch VMs (i.e., "silver" instances), with a different billing policy.
- **Billing depends on performance:** the CSPs may provide different billing policy to clients depending on the performance of the VMs, i.e., lower billing when lower performance is provided because the promised performance guarantees were not met. Depending on the priority of the VM, the thresholds for maximum allowable performance penalty may differ and many thresholds for different levels of performance may exist.

Based on the VMs prioritization and the performance of the VMs, the ACTiCLOUD architecture improves resource efficiency through the metric of gain by selectively placing, remapping, and migrating the VMs across the available hardware resources. Section 3.1 explains the gain metric in more detail.

Target value: We target an increase of resource efficiency through the aforementioned metric of gain of 15%-50% depending on the execution scenario.

Baseline: To demonstrate such improvements, we compare the ACTiCLOUD architecture against a vanilla OpenStack installation that is enhanced with straightforward optimizations for resource efficiency. Note that the comparison takes place on the same hardware platforms and uses the same components where applicable (e.g., guest OS version).

Benchmarks: We use a representative set of benchmarks that span a wide range of application behavior regarding the utilization of shared resources. More specifically, we use benchmarks from Spec2006, SPECjvm2008, and Parsec benchmark suites, and other benchmarks. In addition, we use Neo4j with LDDB and MonetDB with TPC-H.

Testbeds: To demonstrate and evaluate the various components of the ACTiCLOUD architecture, we use testbeds that focus on: (i) a single node, e.g., a single Numascale system or a single Exynos server of the KMAX system, (ii) multiple nodes, e.g., using a cluster of off-the-shelf x86 servers, or using all Exynos servers of the KMAX system, and (iii) multiple sites with each site consisting of multiple nodes.

Correlation with business scenarios and use-cases: This evaluation scenario is directly correlated with the Business Scenario 1 "Effective consolidation for increased revenue and reduced TCO", Business Scenario 4 "Collaboration with sibling cloud sites", Business Scenario 5 "Enhanced dependability and availability", Use Case 1 "Execution of typical cloud applications", Use Case 2 "Database applications with constant workload and intermittent uptime", and Use Case 7 "Enterprise Operations".

2.2 Performance Stability

One of the main reasons that current cloud data centers are suffering from low utilization is that CSPs are conservative when allocating resources because of the interference that might appear among VMs. Such interference introduces performance instability in terms of increased performance variation compared to standalone execution. By intelligently placing, remapping, and migrating the VMs across the hardware resources while avoiding interference, the ACTiCLOUD architecture mitigates performance degradation and instability due to interference, and increases performance stability guarantees for applications.

KPI: Similarly with the previous evaluation scenario, increasing performance stability may be translated to various metrics, such as increasing total system throughput or gain metric (€). In addition, the KPI for performance stability is also based on the aforementioned assumptions of (i) prioritization of the VMs, and (ii) SLAs depending on performance.

In the evaluation of the ACTiCLOUD architecture for performance stability we use the metric of preserving the performance of high-priority (i.e., gold) VMs for the X% of the execution time over a threshold Y% with respect to standalone execution. For example, assuming a gold VM that hosts a large database that serves 100*K transactions per second and runs for 100 minutes when running alone, given X=98% and Y=90% then the metric of performance stability would require for that VM to sustain at least 90*K transactions per second for at least 98 minutes when running on a server with multiple other VMs. Note that the indicator of performance loss depends on each application, e.g., instructions/sec, ops/sec, iops/sec, maximum/average latency, etc.

Target value: For high-priority VMs, we target achieving performance stability for X=95% of execution time over a threshold of Y=80% with respect to standalone execution. While our primary target is enabling performance stability for the high-priority VMs in this evaluation scenario, we also consider enabling performance stability for low-priority VMs using less strict target values when appropriate and in a best-effort approach to increase utilization.

Baseline: To demonstrate such improvements, we compare the ACTiCLOUD architecture against a vanilla OpenStack installation that does not provide any support for isolation or that provides conservative isolation policies leaving unused resources. Note that the comparison takes place on the same hardware platforms and uses the same components where applicable (e.g., guest OS version).

Benchmarks: We use a representative set of benchmarks that span a wide range of application behavior regarding the utilization of shared resources. More specifically, we use benchmarks from Spec2006, SPECjvm2008, and Parsec benchmark suites, and other benchmarks. In addition, we use Neo4j with LDBC and MonetDB with TPC-H.

Testbeds: To demonstrate and evaluate the various components of the ACTiCLOUD architecture, we use testbeds that focus on: (i) a single node, e.g., a single Numascale system or a single Exynos server of the KMAX system, and (ii) multiple nodes, e.g., using an entire cluster of off-the-shelf x86 servers, or using all Exynos servers of the KMAX system, and (iii) multiple sites with each site consisting of multiple nodes.

Correlation with business scenarios and use-cases: This evaluation scenario is directly correlated with Business Scenario 2 "Workload prioritization", Use Case 1 "Execution of typical cloud applications", and Use Case 3 "Database applications with constant workload and continuous uptime".

2.3 Scalability in Resource Provisioning

Current cloud data centers face challenges to support resource demanding applications, such as large in-memory databases, because the provided resources are bounded by the physical resources that are present on a single server. ACTiCLOUD delivers a unified pool of resources by aggregating resources from different physical machines as compared to those provided by state-of-the-art offerings in typical cloud installations.

KPI: Increased scalability in resource provisioning translates to enabling the execution of a broader and challenging class of applications in the cloud. This in turn may translate to increased gain from the CSPs' perspective and to reduced Total Cost of Ownership (TCO) from the users' perspective. To evaluate ACTiCLOUD's offering for resource scalability, we use the metric of largest possible size of dataset that can be executed in an ACTiCLOUD-enabled cloud data center under a specified performance metric.

Target: We target an increase of 10x regarding processing and analyzing in in-memory fashion the largest possible size of the dataset while incurring up to 2x slowdown in terms of throughput (e.g., requests/transactions per second), when compared to processing a smaller dataset that uses all available resources on a conventional infrastructure. Note that the slowdown in throughput could be due to accessing data that are allocated in remote NUMA nodes in a large NUMA system, such as the Numascale system. So the throughput of transactions could be lower compared to when processing a smaller dataset on a commodity system, but the processing and analysis of significantly largest datasets with acceptable performance becomes possible.

Baseline: To demonstrate such improvements, we compare the execution of in-memory databases that run on the aggregated resources that ACTiCLOUD architecture offers, against in-memory databases that are limited by the physical resources that are present on commodity off-the-shelf servers or against disk-based databases.

Benchmarks: We use database benchmarks, i.e., TPC-H and Air Traffic with MonetDB, and LDBC with Neo4j. We also use benchmarks from the HPC domain.

Testbeds: To demonstrate and evaluate the various components of the ACTiCLOUD architecture regarding scalability in resource provisioning, we use as testbed the Numascale system.

Correlation with business scenarios and use-cases: This evaluation scenario is directly correlated with Business Scenario 3 "Hosting larger workloads", Use Case 1 "Execution of typical cloud applications", Use Case 4 "Database applications with predictable workload burst", and Use Case 6 "Analysis of social networks with high dynamism".

2.4 Elasticity in Resource Provisioning

Current cloud data centers face challenges to support elasticity regarding resource provisioning for resource demanding applications, such as large in-memory databases. Resource scalability

targets the ability to run fast on a scaled up or scaled down server. ACTiCLOUD leverages the components that provide scalability of resources and enables cloud systems to provision and deprovision high amounts of resources in an elastic way based on application requirements.

KPI: Enabling elasticity in resource provisioning in the cloud requires the efficient execution of a broad and challenging class of applications which require resources on demand depending on workload or time. This in turn may translate to increased revenue from the CSPs' perspective and to reduced TCO from the users' perspective.

The objective is to address elastic provisioning needs of rapidly changing requests while preserving their performance levels. In order to evaluate the elasticity of resources, we will use (i) the classification metric of whether elasticity is supported, (ii) the throughput metric as described in Section 2.1 that translates to benefits for both the CSPs and users in terms of utilization, and (ii) the performance stability metric as described in Section 2.2 for showing that the right amount of resources have been allocated to provide elasticity.

Target: We aim to scale resources elastically to respond to application requirements while minimizing the necessary resources to achieve that, in addition to providing increased resource efficiency, performance stability, and support for scalability as explained earlier. Specifically, ACTiCLOUD provides the mechanisms for applications to use different OpenStack flavors, i.e., VMs of different characteristics in terms of cores, memory, etc., during their lifetime depending on their demand, combined with the ACTiCLOUD logic for resource efficiency and stability. This is especially important for applications with variable workloads during their lifetime.

Baseline: To demonstrate such improvements, we compare the execution of benchmarks that have variable resource demands during their lifetime and benefit from resource-aware scale-up/down features against a vanilla OpenStack installation with no scale-up/down support and with resource-unaware scale-up/down support.

Benchmarks: We primarily use microbenchmarks with application specific metrics and MonetDB with TPC-H to showcase the support for elasticity that ACTiCLOUD provides.

Testbeds: To demonstrate and evaluate the various components of the ACTiCLOUD architecture, we use testbeds that focus on: (i) a single node, e.g., a single Numascale system or a single Exynos server of the KMAX system, and (ii) multiple nodes, e.g., combining the Numascale system with other off-the-shelf x86 servers, or using entirely a cluster of x86 servers, or using all Exynos servers of the KMAX system.

Correlation with business scenarios and use-cases: This evaluation scenario is directly correlated with Use Case 1 "Execution of typical cloud applications", Use Case 4 "Database applications with predictable workload burst", Use Case 5 "Applications with unpredictable workload burst", Use Case 6 "Analysis of social networks with high dynamism", and Use Case 7 "Enterprise Operations".

3 Resource efficiency

In this section we evaluate how the ACTiCLOUD architecture improves resource efficiency. We particularly focus on ACTiManager and the MicroVisor.

3.1 ACTiManager

ACTiManager is ACTiCLOUD's resource manager for cloud computing. Deliverable D2.4 describes the final version of the 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 the OpenStack cloud management platform.

3.1.1 Methodology

We describe here the methodology that we use to evaluate ACTiManager. Note that we use the same methodology for evaluating ACTiManager in improving performance stability in Section 4.

Infrastructure

We implement ACTiManager as a pluggable framework interfacing with: (i) OpenStack Pike for creating and migrating VMs among servers, (ii) the libvirt fine-grained management tool for (re)pinning cores and memory within servers, and (iii) the QEMU/KVM hypervisor layer.

We setup Openstack Pike on a cluster of four Intel x86-64 dual-socket servers. Each server is equipped with two Intel Xeon CPU E5-2630 v4 processors with 10 cores for a total of 20 physical cores on each server and 80 physical cores on the whole cluster (hyperthreading is disabled). The processors run at a fixed clock speed of 2.20GHz. The physical cores of each processor have private L1 and L2 caches of sizes 32KB and 256KB respectively, and they all share an L3 cache of size 25MB. The servers run Ubuntu Linux 18.04 with kernel version 4.15.0. For the virtualization layer we use libvirt 4.0.0 with QEMU/KVM 2.11.1. One server acts both as the Openstack controller node and as a compute node. The other three servers act only as compute nodes. Note that we decide to use this testbed setup because we need to use a testbed that consists of multiple NUMA systems, in order to showcase the end-to-end functionality of all the components of ACTiManager. Using the ACTiCLOUD hardware platforms, i.e., Numascale and KMAX systems, would limit us in evaluating only the internal component (in the setup of a single Numascale or a single Exynos of KMAX) or only the external component (in the setup of using multiple Exynos's of KMAX). We indeed use the ACTiCLOUD hardware platforms to evaluate specific components of ACTiManager in Sections 4 and 6.

Pricing Model

Section 3 of Deliverable D2.4 "Distributed Cloud Resource Manager v2.0" describes in detail the pricing model that drives the decision making logic of ACTiManager. We include that information here to make this document self-contained.

ACTiManager differentiates from previous research resource managers on the target function it applies to guide its decision making. Previous resource managers distinguish only between latency critical and best effort applications, and targeting 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. Because the co-execution of VMs introduces interference, we define different levels of interference when co-executing multiple workloads for

gold and silver VMs. 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 an (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 logic of ACTiManager.

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:

$$\begin{aligned} \text{price}_{\text{gold}} &= m_g \text{ if } S \leq t_g \\ \text{price}_{\text{gold}} &= 0 \text{ if } S > t_g \end{aligned}$$

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 varieties. The first type 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):

$$\begin{aligned} \text{price}_{\text{silver}} &= m_s \text{ if } S \leq t_s \\ \text{price}_{\text{silver}} &= 0 \text{ if } S > t_s \end{aligned}$$

The second variety 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 a roughly constant amount for the execution of their job. This is captured by a pricing formula as follows:

$$\begin{aligned} \text{price}_{\text{silver}} &= m_s / S \text{ if } S \leq t_s \\ \text{price}_{\text{silver}} &= 0 \text{ if } S > t_s \end{aligned}$$

Figure 3.1 summarizes the three models that we use in our decision making within ACTiManager. 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.

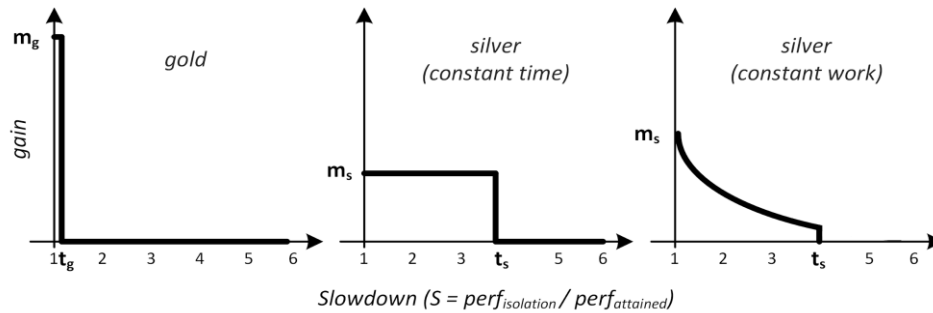


Figure 3.1: Pricing models for gold and silver applications.

We also consider that turning on a server in a cloud datacenter is associated with a specific monetary cost (m_{server} in monetary units per time unit) that is defined by the cloud provider and is related to the 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 implementation, we consider that there is a cost to turn on a NUMA socket of the server (m_{socket}), but can be trivially extended to support more detailed power management schemes (e.g., turning on/off parts of the socket).

Benchmarking

For our evaluation we run a set of experiments each lasting three hours with the following configuration:

- In the first two hours we feed the datacenter with a sequence of VMs at such a rate as to maintain the load (in terms of number of vCPUs currently executing) to a predefined level (depending on the load scenario as explained next). In case a VM cannot start execution (e.g., due to lack of available resources), we stop spawning new VMs until the blocked one is accepted. Hence, the total number of accepted/executed VMs differs among different resource management schemes. In the last hour of the experiment we stop spawning new VMs and wait for the running VMs to finish their execution.
- The sequence of the VMs is generated before the execution as a queue with 200 VMs and is the same across executions with different resource management schemes. Each VM has an arrival timestamp indicating the point in the execution timeline when it arrives in the queue of the datacenter.
- We employ four different VM types with 1, 2, 4, and 8 vCPUs respectively and 2GB of RAM per vCPU. We determine the fraction of each flavor based on public traces from Microsoft's Azure cloud¹. More specifically in our experiments we have 50%, 25%, 15% and 10% of VMs with 1, 2, 4 and 8 vCPUs, respectively.
- The VMs use an Ubuntu 18.04 image and each executes one of the 16 SPEC 2006 benchmarks² listed in Table 3.1. VMs with more than 1 vCPU execute one instance of the specific benchmark on each vCPU.

¹ 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>

² John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. SIGARCH Comput. Archit. News 34, 4 (Sept. 2006), 1–17. <https://doi.org/10.1145/1186736.1186737>

- The execution time of each VM ranges between 15 and 45 minutes when run in isolation. We vary the execution time by executing the same benchmark as many times as necessary to reach the total execution time that we target within the same VM. After each execution of the internal SPEC benchmark, the VM reports its performance and we collect these performance numbers after the end of the execution to calculate the slowdown and the actual profit of the datacenter.

Table 3.1: The SPEC 2006 benchmarks used in our experimental evaluation and their interference classification.

Benchmark	Noisy	Quiet	Sensitive	Insensitive
astar	✓		✓	
bwaves	✓			✓
bzip2		✓	✓	
cactusADM	✓		✓	
calculix		✓		✓
gamess		✓		✓
GemsFDTD	✓			✓
gobmk		✓		✓
leslie3d	✓			✓
libquantum	✓		✓	
mcf	✓		✓	
milc	✓		✓	
namd		✓		✓
omnetpp	✓		✓	
povray		✓		✓
tonto		✓		✓

Table 3.2: Pricing policy parameters. In all experiments we use $t_g = 1.2$ and $t_s = 4$.

	pricing 1	pricing 2	pricing 3
m_g	10	5	10
m_s	1	1	1
m_{server}	20	10	0
m_{socket}	5	2	0

Benchmarking Parameters

We evaluate the different resource management schemes under diverse scenarios by varying the following parameters in our experiments:

- *Datacenter load*: We use three different loads, i.e., low, medium and high when 37.5%, 75% and 150% of the cores of our cluster are requested, respectively.
- *Gold/Silver VMs*: We use four different fractions of gold VMs: 0%, 20%, 50% and 100%.
- *Noisy/Quiet VMs*: We use three different fractions of noisy VMs: 20%, 50% and 80%.
- *Pricing model*: Since ACTiManager considers the pricing policy of the datacenter in its decision making, we evaluate it with the three pricing policies presented in Table 3.2. Pricing 1 and 2 test a different charging balance between gold and silver and pricing 3 tests the behavior of ACTiManager when the cost of compute resources is discarded.

We label each configuration as "*load_%gold_%interference*", e.g., an experiment with medium load, 50% gold VMs and 20% noisy is labeled as "*medium_50_20*". Unless otherwise specified, the pricing model is the pricing 1 of Table 2.

Since covering the entire parameter space would lead to huge measurement times (more than two months), we select some meaningful configurations - in terms of total load, percentage of gold VMs and percentage of noisy VMs - of this search space with a goal to capture practical cases and corner cases that can shed further light to the operation of each policy. Also, to keep the experiment times low we limited the execution of each setup to 3 hours and the time of each benchmark to 15-45 mins, violating in this way our assumption that applications are long-running. For this reason, we have excluded the time needed to characterize the benchmarks within the laboratory node, as we believe that this would lead to an unfair estimation of resource overheads for ACTiManager.

Resource Management Schemes

We evaluate the following resource management schemes:

- Openstack (vanilla) is the default resource management policy that incorporates neither prioritization (disregards gold/silver) nor interference awareness. It serves as the most primitive baseline, and also validates that the overheads of the sophistication of ACTiManager do not outweigh any benefits of prioritization and interference awareness. This policy accepts a new VM as long as a specific threshold for oversubscription is not crossed. In our case we set this threshold to $t_s = 4$, the slowdown accepted by silver VMs in our setup.
- GNO (Gold Not-Oversubscribed) is the straightforward prioritization policy that assigns one core per vCPU for the gold VMs and an over-subscription threshold $t_s = 4$ for the silver VMs, again. GNO is interference unaware.
- Socket is both prioritization and interference-aware and applies the straightforward isolation policy, i.e. each gold VM is executed in an entire socket in isolation. Silver VMs are handled as in the previous policies.
- ACTiManager_no_mit applies only interference avoidance (no mitigation).
- ACTiManager is the full resource manager.

3.1.2 Results

We assess the behavior of each policy under eight different configurations. Qualitatively, we are interested in three metrics: (i) how much load (throughput) each policy accommodates in its execution, (ii) how many compute resources were consumed, and (ii) how much of this load executed respecting its SLA (i.e., performance close to standalone execution). In this section we

focus on the first two metrics that target resource efficiency, together with the gain metric from the CSP point of view as described earlier, while in Section 4 we focus on the third metric that targets performance stability.

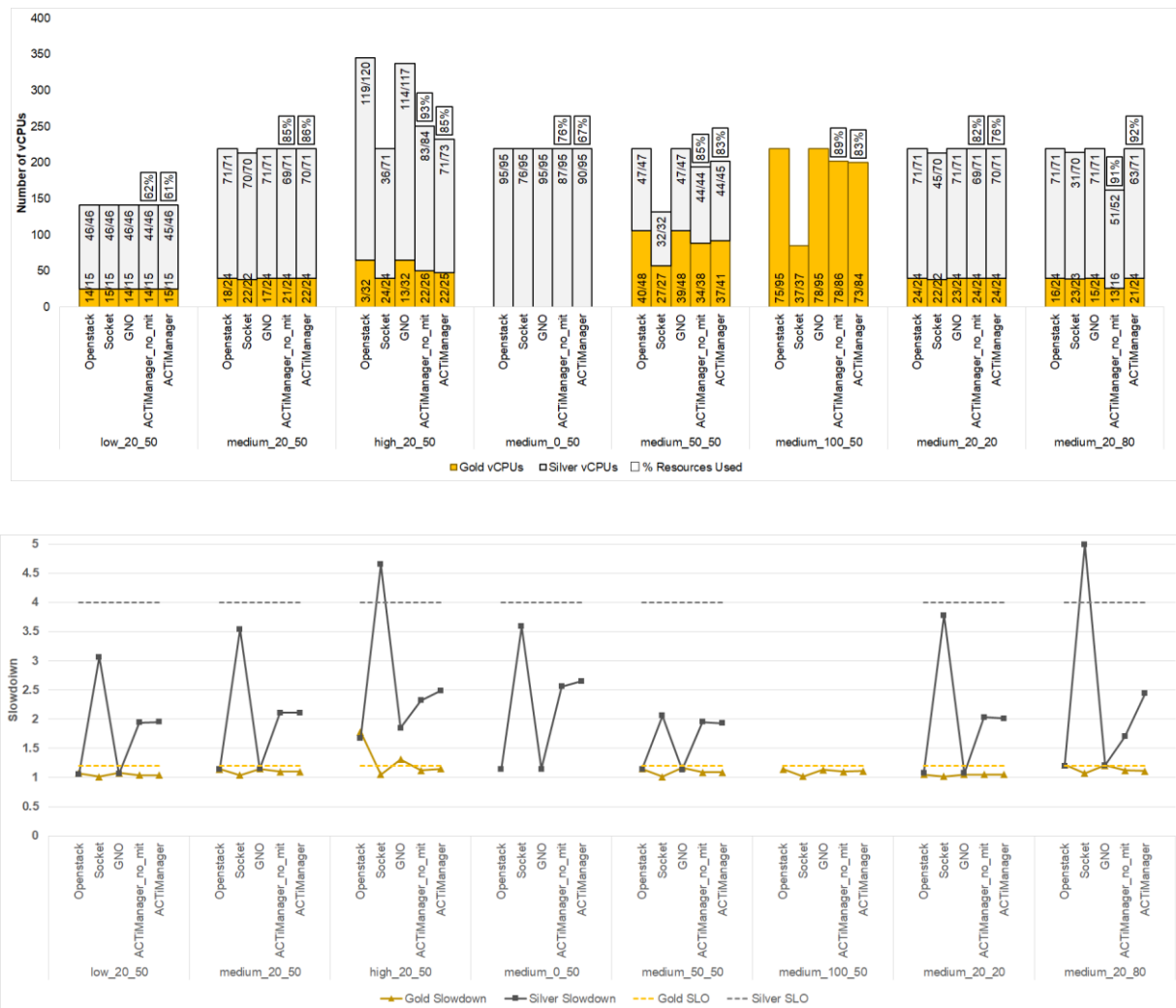


Figure 3.2: Datacenter throughput (a) and application slowdown (b). The top figure (a) shows the throughput, i.e., the total number of all accepted VMs' vCPUs, broken down into gold and silver vCPUs. Numbers within the bars designate the number of gold/silver VMs that respected their SLAs. The number in the rectangle shows the percentage of the compute resources utilized (shown only for ACTiManager as other policies use 100% of resources). The bottom figure (b) shows the average slowdown of applications.

Figure 3.2a demonstrates the throughput of each policy (i.e., the total number of all VMs' vCPUs accepted for execution by each policy). Within each bar we present the number of VMs that respected their SLAs and the total number of VMs for each class that were executed (gold, silver). Regarding resource utilization, since we did not incorporate a consolidation policy for Openstack, GNO and Socket; these strategies consume 100% of the compute resources. Figure 3.2b shows the average slowdown for each category of VMs (i.e., gold/silver).

low_20_50: In this configuration with low load, clearly there is limited need for a sophisticated resource allocation policy. Nevertheless, we observe that ACTiManager drastically reduces the resources needed to 61%, but at the cost of one silver and one gold VM violating their SLA as ACTiManager tries to co-locate as many workloads as possible using less resources.

medium_20_50: This scenario starts to put pressure on the resource managers. We observe that OpenStack and GNO penalize the execution of gold VMs that are suffering from interference, while Socket severely penalizes silver VMs since it largely oversubscribes them, also disregarding any impact from interference. On the other hand, ACTiManager keeps the QoS high by taking interference into account when co-locating VMs on nearby resources (servers, sockets, and cores) and saves some non-negligible part of the resources as it uses 86% of the available resources.

high_20_50: Under high load, Openstack (as expected) and GNO perform poorly as they are not able to keep the QoS for the majority of gold VMs, also paying their aggressiveness to accept a large number of VMs. Socket also penalizes the execution of silver VMs, rendering their execution in the system highly problematic. ACTiManager keeps a very good balance between throughput, application performance and resource utilization (85%).

medium_0_50: This scenario involves the execution of only silver VMs. In this scenario all policies operate similarly. ACTiManager limits the used resources (67%), because of the assumption that silver VMs can be oversubscribed on the same cores, but pays a small penalty in QoS. This demonstrates that we need to better fine-tune our scheme making it less aggressive in resource savings.

medium_50_50: When increasing the percentage of gold VMs in the mix, Openstack and GNO face problems in keeping their QoS, while Socket becomes more conservative to deal with this load. ACTiManager, again, operates in a more balanced way using 83% of the available resources.

medium_100_50: In the configuration with only gold VMs, Openstack and GNO again face problems due to interference and Socket becomes highly conservative in accepting VMs and allocating resources, being able, though, to have a perfect QoS for the applications (as expected). ACTiManager approximates the aggressiveness of OpenStack and GNO, keeps the QoS high, but pays some cost in SLA violations while using 83% of the available resources.

medium_20_20: In this configuration Socket penalizes severely the silver VMs. Openstack, GNO and ACTiManager behave similarly, with ACTiManager using 76% of the available resources.

medium_20_80: When putting pressure with a more contentious and sensitive configuration, OpenStack and GNO penalize the performance of gold VMs. Socket keeps the QoS of gold VMs but fails to preserve the QoS of silver VMs. ACTiManager is able to operate on a better point, with a limited number of SLA violations for gold and silver VMs, using 92% of the available resource.

Overall, ACTiManager presents a much more balanced operation across the various configurations where it demonstrates either the best or close to the best behavior in terms of throughput and application performance while requiring less resources, leading to increased resource efficiency. Furthermore, the operation of ACTiManager performs consistently well for all configurations, which is not the case for the rest of the schemes. Still our results demonstrate that there is room for improvement in ACTiManager towards an even better balance between the number of accepted VMs, respected SLAs, and used resources. Finally, although there is not a notable difference between the two versions of ACTiManager we present, interference mitigation has the potential to slightly improve the behavior of ACTiManager in some cases, while being slightly more conservative in others.

Datacenter profit results

For this set of results, we turn our attention to the side of the CSP and assess the profit gained from each of the considered resource allocation schemes. We use the three different pricing schemes presented in Table 3.2. Note that in Section 2.1 where we described our evaluation

strategy for resource efficiency, we used this metric with a target value of 15%-50% increased gain, depending on the execution scenario.

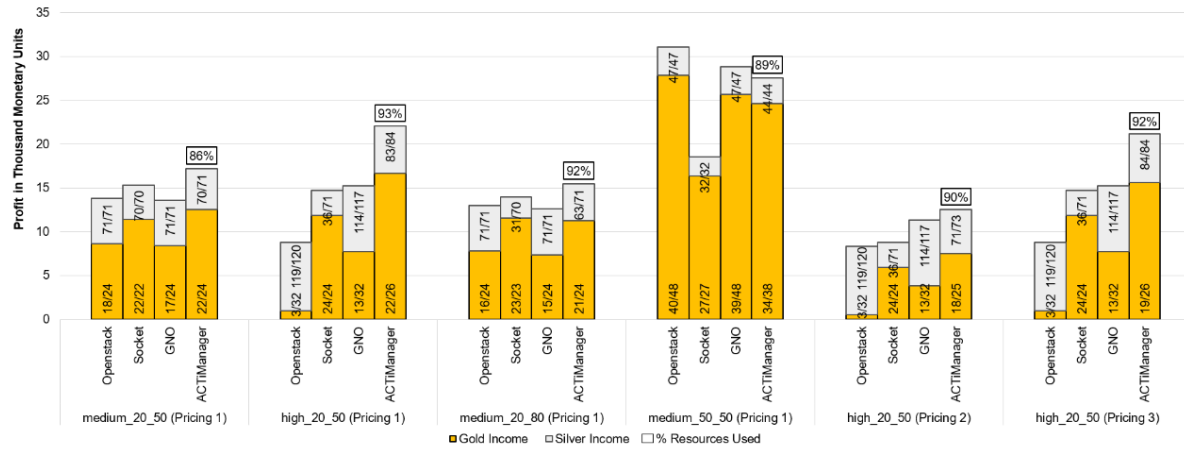


Figure 3.3: Profit per resource management scheme. The numbers inside the bars designate the number of gold and silver VMs that respected their SLAs. The numbers in the rectangles show the percentage of the compute resources utilized.

Figure 3.3 presents the profit in monetary units for the cloud service provider under six scenarios of medium and high load. We observe that in five out of the six scenarios, ACTiManager is able to provide significant profit increases that range from 12% to 49%. For the medium_50_50 scenario, Openstack and GNO outperform ACTiManager with Openstack achieving a 13% increase compared to our policy. ACTiManager could be improved on that front by co-locating VMs less aggressively and using more resources. Note, however, that Openstack and GNO as also discussed previously, are not able to keep a competitive performance under scenarios with high load, whereas ACTiManager can with usually less resources.

3.1.3 Summary

Overall we observe that ACTiManager increases resource efficiency in terms of both: (i) performed work with respect to used resources and (ii) profit from the CSP point of view by properly placing, remapping, and migrating the VMs across the hardware resources.

3.2 HyperVisor

As discussed extensively in Deliverables D2.3 and D4.4: “ACTiCLOUD Final Prototype”, the Hypervisor (HV) or virtual machine monitor (VMM) is a system software layer that creates, manages, and runs Virtual Machines (VMs). In the ACTiCLOUD architecture the Hypervisor layer presents the guest operating systems (OS) in the VMs with a virtual operating platform and manages the execution of the guest operating systems. The VM abstractions created by the hypervisor provide a secure environment, isolating user tasks from other tasks, applications, and other systems on the network. Tasks in this case entail the computation that takes place within an application as well as within the system kernel, so the hypervisor ensures isolation and security at both the application and operating system kernel levels. By providing isolation, the hypervisor layer enables the co-location of different applications and their operating systems on the same hardware (i.e., multi-tenancy), which essentially enables the operation of cloud computing, both on the premises of an organization (private cloud), or flexible leasing of virtual machines on Cloud Service Providers (CSP) (public cloud).

In this section we present the final evaluation of the hypervisor layer, which in the ACTiCLOUD architecture is the MicroVisor platform, over two different hardware architectures, large Intel x86 servers (for scale-up) and KMAX (for scale-out). As also reported in D4.3, the objective of the rack-scale MicroVisor platform, through the efficient hypervisor layer and the OpenStack integration, is to progress state-of-the-art cloud management approaches with mechanisms to:

- Drastically increase the resource efficiency of cloud infrastructures in terms of throughput per resource unit (*Strategic Objective S01.1 on resource efficiency*).
- Provide applications with better performance by lowering virtualization overheads and strict performance guarantees through better resource allocation and placement (*Strategic Objective S01.2 on performance stability*).

Additionally, the rack-scale MicroVisor supports all the necessary mechanisms for the deployment and management of scale-up and scale-out resources offered by the underlying hardware (*Strategic Objective S02.1 on scalability in resource provisioning*).

Furthermore, interacting with the ACTiManager and the applications running in Virtual Machines (VMs), the MicroVisor allows placement and scheduling of VM resources on the hardware and responds to dynamic resource requests of the ACTiManager to address changes in application demand (*Strategic Objective S02.2 on elasticity in resource provisioning*). Further information regarding the interaction of the ACTiManager with the MicroVisor is reported in D4.4: “ACTiCLOUD Final Prototype”.

Finally, the rack-scale MicroVisor is an essential component for the realization of ACTiCLOUD’s business scenarios (further discussed in deliverables D1.1 and D1.2):

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

The evaluation of the MicroVisor (hypervisor layer) is analyzed in this section.

3.2.1 Benchmarks and Metrics

The main benchmark used in this evaluation is Sysbench³, which was also used during the intermediate evaluation phase (see D4.3). Using the same benchmark in both evaluation phases enables us to directly compare the results. Sysbench is one of the most popular benchmarking utilities used to test Virtual Machine and Cloud Server performance, providing easy to find and compare performance results, since it is widely used worldwide.

SysBench, as also reported in D4.3, is a modular, cross-platform and multi-threaded benchmark tool for evaluating OS parameters that are important for a system running a database under intensive load. The idea of this benchmark suite is to quickly get an impression about overall system performance without setting up complex database benchmarks or even without installing a database at all.

SysBench features allow for testing of the following system properties:

- Scheduler performance
- Memory allocation and transfer speed
- POSIX threads implementation performance
- File I/O performance

³ <https://wiki.mikejung.biz/Sysbench>

- Database server performance (requires DB setup)

SysBench spawns a number of threads, specified by the user, which execute requests in parallel. The actual workload generating requests depends on the specified test mode. Test modes are implemented by compiled-in modules and may have additional (or workload-specific) options.

We use SysBench to measure all the aforementioned system properties. However, for brevity reasons, we report only the most representative performance results, using the following workloads.

CPU workload

When running with the CPU workload, SysBench verifies prime numbers by doing standard division of the number by all numbers between 2 and the square root of the number. If any number gives a remainder of 0, the next number is calculated. As expected, this workload stresses the CPU cores in specific areas (e.g., integer and floating point operations, branching, etc.).

The benchmark can be configured with the number of simultaneous threads and the maximum number to verify if it is a prime. The number of events per second is used as a performance indicator. We have used the following parameters in the cpu workload:

```
# sysbench --test=cpu --cpu-max-prime=1000000 --threads=X
```

Memory workload

When using the memory test in SysBench, the benchmark allocates a memory buffer and reads or writes to it, each time for the size of a pointer (32bit or 64bit), and for each execution until the total buffer size has been read from or written to. This is repeated until the provided memory volume (size) is reached. Users can also provide the number of threads, different sizes in memory block buffer and the type of requests (read or write, sequential or random). The performance metric obtained using the memory workload (given the same parameters) is the throughput and operations per second.

OLTP / MySQL workload

This is a database/MySQL performance test which allows an estimate of MySQL performance in specific scenarios. It can simulate read-only workloads (e.g., using only SELECT queries), or test mixed read and write workloads. For this benchmark we have set up the standard MySQL 5.7 for the OLTP runs, and we measure performance in transactions per second, for each scenario.

Additional Benchmarks

Besides SysBench, we also evaluate the MicroVisor with the following benchmarks/tools:

- The Unixbench⁴ benchmark, measuring basic system performance (OS and CPU). UnixBench is a common tool used to test and compare the performance of servers or VMs provided by different vendors. The purpose of UnixBench is to provide a basic indicator of the performance of a Unix-like system; hence, multiple tests are used to test various aspects of the system's performance. These test results are then compared to the scores from a baseline system to produce an index value, which is generally easier to handle than the raw scores. UnixBench tests consist of two parts: single-process tests and multi-process tests. UnixBench consists of a number of individual tests that are targeted at specific areas.

⁴ <https://github.com/kdlucas/byte-unixbench>

- The fio⁵ (Flexible I/O) tester, measuring storage I/O and filesystem performance.
- The iperf⁶ tool measuring TCP network performance.
- The stream⁷ benchmark measuring memory bandwidth.

For brevity reasons we only report representative SysBench and Unixbench results in this document.

3.2.2 Evaluation of the MicroVisor on large Intel x86 servers

The MicroVisor platform can be deployed on on-premise infrastructures to run workloads locally. A MicroVisor cluster is comprised of a minimum of 2 nodes and is fully connected to the network environment of the infrastructure, such as the VLAN setup which could be used by the instances running on the MicroVisor platform. The following hardware configuration per node is used in the performance benchmarking reported for ACTiCLOUD:

Name	RAM	Physical CPUs	Storage	Network
Intel BP (Bobcat Peak) ⁸	192GB	2 x Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz 10 cores	2 x 2.0TB NVMe direct attached	20 Gbps (2 x 10 GBps NICs)

The MicroVisor platform focuses on removing the overhead of device IO processing for a guest VM. By directly mapping the host side paravirtual IO support logic in the VMM layer the MicroVisor platform can remove a significant part of the overhead incurred in virtualising IO access to physical resources. Figures 3.4 and 3.5 show the results which illustrate the benefits of this architectural approach for raw measurements of inter-VM TCP network throughput and latency measured on the same hardware for MicroVisor(MV) vs KVM technology. Figure 3.4 shows the throughput as measured between two VMs on the same host for KVM vs MicroVisor, where we observe that the MV throughput is 21% faster.

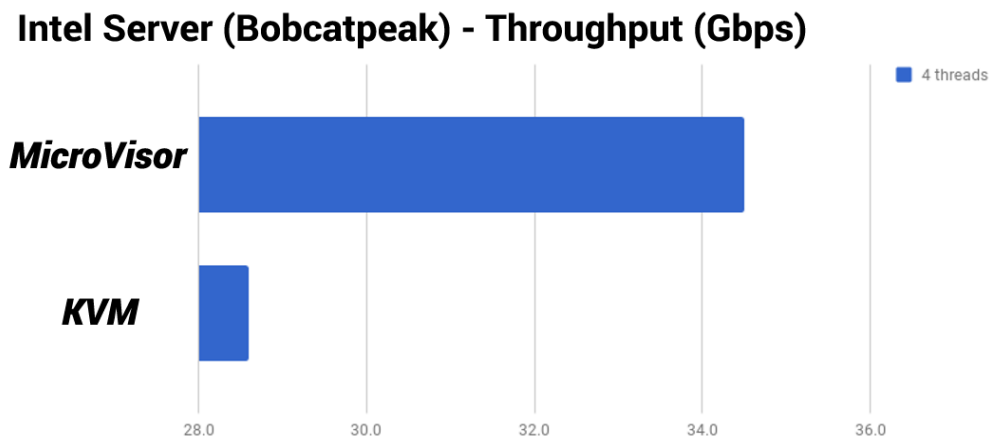


Figure 3.4: The results of inter-VM network throughput for MicroVisor (MV) and KVM.

⁵ https://www.storagereview.com/fio_flexible_i_o_tester_synthetic_benchmark

⁶ <https://iperf.fr/>

⁷ <https://www.cs.virginia.edu/stream/>

⁸ <https://ark.intel.com/content/www/us/en/ark/products/codename/59838/bobcat-peak.html>

Figure 3.5 depicts the latency as measured again between two VMs on the same host for KVM vs MicroVisor, where the MV latency is 58% lower.

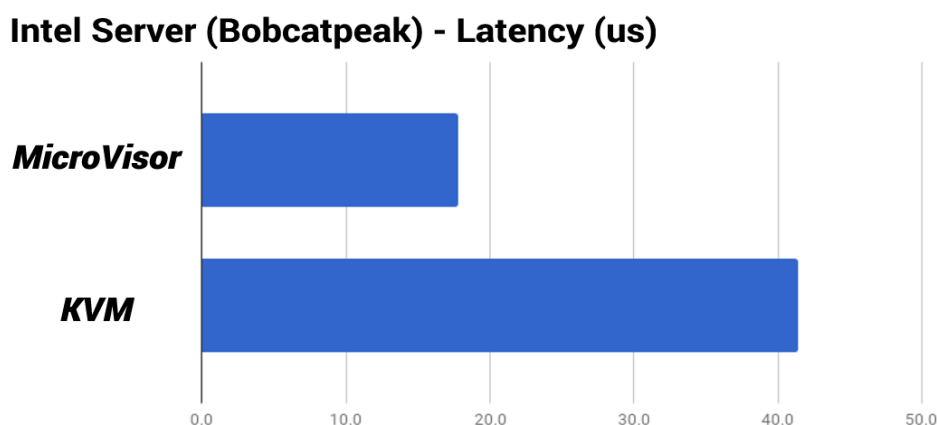


Figure 3.5: The results of inter-VM network latency for MicroVisor (MV) and KVM.

CPU performance benchmarks

In order to validate the performance of VMs running on the MicroVisor platform, UnixBench is selected to measure and compare the operating system running when run in a VM against when run on bare-metal. The results depend not only on the hardware used, but also on the operating system, libraries, and even the compiler. Thus, the same hardware (Intel Bobcat peak servers) and operating system (Ubuntu server 18.04 LTS) are used for the tests in all three setups:

- The operating system is running directly on the bare-metal machine (i.e., no hypervisor).
- The operating system is running on a VM on the MicroVisor platform.
- The operating system is running on the KVM hypervisor.

Unixbench performs the following benchmark tests⁹:

- **Dhrystone** – Used to measure and compare the CPU performance in general - measured in loops per second (lps).
- **Whetstone** – Used to measure the speed and efficiency of floating-point operations - measured in Millions of Whetstone Instructions Per Second (MWIPS).
- **Execl Throughput** – Used to measure the number of execl calls that can be performed per second - measured in loops per second (lps).
- **File Copy** – Used to measure the rate at which data can be transferred from one file to another - measured in kilobits per second (Kbps).
- **Pipe Throughput** – Used to measure the number of times a process can write 512 bytes to a pipe and read them back - measured in loops per second (lps).
- **Pipe-based Context Switching** – Used to measure the number of times two processes can exchange an increasing integer through a pipe - measured in loops per second (lps).
- **Process Creation** – Used to measure the number of times a process can fork and reap a child that immediately exits - measured in loops per second (lps).
- **Shell Scripts** – Used to measure the number of times a process can start and reap a set of eight concurrent copies of a shell script, where the shell script applies a series of transformation to a data file - measured in loops per minute (lpm).
- **System Call Overhead** – Used to estimate the cost of entering and leaving the operating system kernel - measured in loops per second (lps).

⁹ <https://www.ostechnix.com/unixbench-benchmark-suite-unix-like-systems/>

Individual Test	Bare Metal	KVM	MicroVisor VM
Dhrystone 2 using register variables	3266.4 lps	3252.6 lps	3159.1 lps
Double-Precision Whetstone	837.1 MWIPS	853.4 MWIPS	875.9 MWIPS
Execl Throughput	1168.6 lps	997.7 lps	1193.3 lps
File Copy 1024 bufsize 2000 maxblocks	1579.7 KBps	1633 KBps	1960 KBps
File Copy 256 bufsize 500 maxblocks	989.1 KBps	1003.7 KBps	1229 KBps
File Copy 4096 bufsize 8000 maxblocks	2936.3 KBps	2848.4KBps	3344 KBps
Pipe Throughput	661.9 lps	667.4 lps	839.7 lps
Pipe-based Context Switching	322.9 lps	511.1 lps	582.3 lps
Process Creation	189 lps	883.7 lps	1160.4 lps
Shell Scripts (1 concurrent)	806.7 lpm	2004.5 lpm	2307.3 lpm
Shell Scripts (8 concurrent)	3340.2 lpm	1861 lpm	2134.5 lpm
System Call Overhead	337.7 lps	329.2 lps	503 lps

The CPU performance results reported in the above table are depicted in the following graph:

System performance measured by UnixBench - Baremetal vs. KVM vs. MicroVisor

Hardware: Intel BP Xeon Silver 4114 @2.2GHz VM flavor: 1 vCPU, 2GB RAM, 20GB disk

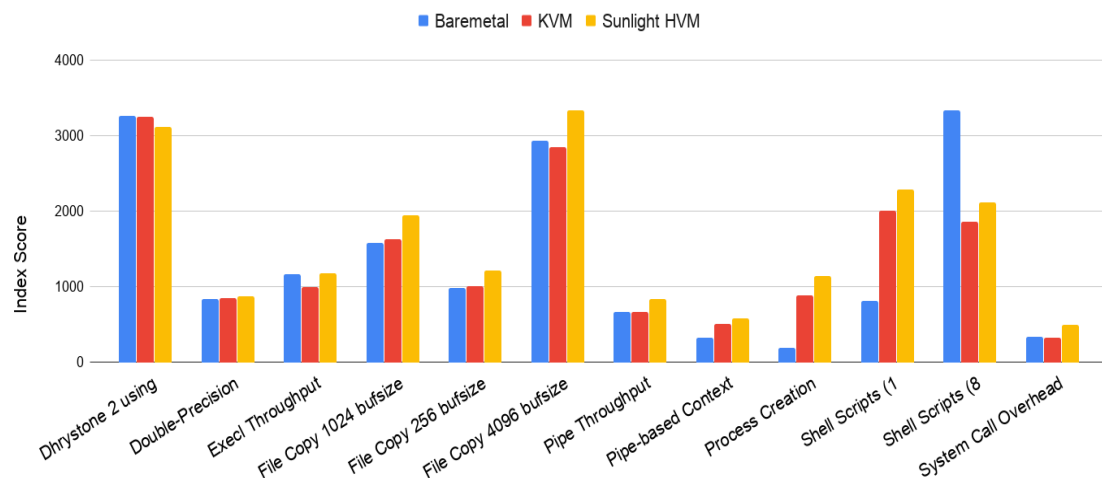


Figure 3.6: UnixBench system performance measurements.

Memory performance benchmarks

The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. To make sure the comparison between bare metal and HVM¹⁰ guest on MicroVisor is fair, the Ubuntu 18.04 stock kernel [4.15.0-45-generic] is selected to be used for STREAM benchmarking in the same hardware system. The configuration of STREAM in this benchmark is as follows:

```
-----
STREAM version $Revision: 5.9 $
-----
```

```
This system uses 8 bytes per DOUBLE PRECISION word.
-----
```

```
Array size = 6291456, Offset = 0
```

```
Total memory required = 144.0 MB.
```

```
Each test is run 1000 times, but only the *best* time for each is used.
-----
```

```
Number of Threads requested = 10
```

The STREAM benchmark results for both bare-metal and runs inside a VM are shown in the tables below:

Bare metal performance

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy	12271.9	0.013149	0.013038	0.013332
Scale	11443.2	0.014221	0.013982	0.014557
Add	12600.4	0.019124	0.019047	0.019210
Triad	12570.7	0.019215	0.019092	0.019381

VM on KVM HyperVisor

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy	11795.0	0.013752	0.013565	0.013842
Scale	10652.4	0.015234	0.015020	0.015441
Add	12288.9	0.020872	0.019530	0.029993
Triad	12223.7	0.020073	0.019634	0.021440

VM on MicroVisor (ParaVirtualized (PV))

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy	11878.3	0.013573	0.013470	0.013752
Scale	11392.0	0.014298	0.014045	0.014529
Add	12171.1	0.019890	0.019719	0.020216
Triad	12168.6	0.019928	0.019723	0.020327

¹⁰ Fully virtualized machines (HVM) require hardware assisted virtualization support (Intel VT-x, AMD AMD-V), whereas Paravirtualized (PV) VMs do not, but can't run operating systems without PV support (you can't run Windows on PV).

VM on MicroVisor (HVM)

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy	11898.7	0.013581	0.013447	0.013783
Scale	11301.8	0.014334	0.014157	0.014631
Add	12308.3	0.019741	0.019499	0.020047
Triad	12241.8	0.019802	0.019605	0.020019

The STREAM memory bandwidth results in the above tables are depicted in the following plot:

Memory bandwidth measured by STREAM

VM flavor: 1 vCPU, 2GB RAM, 20GB disk

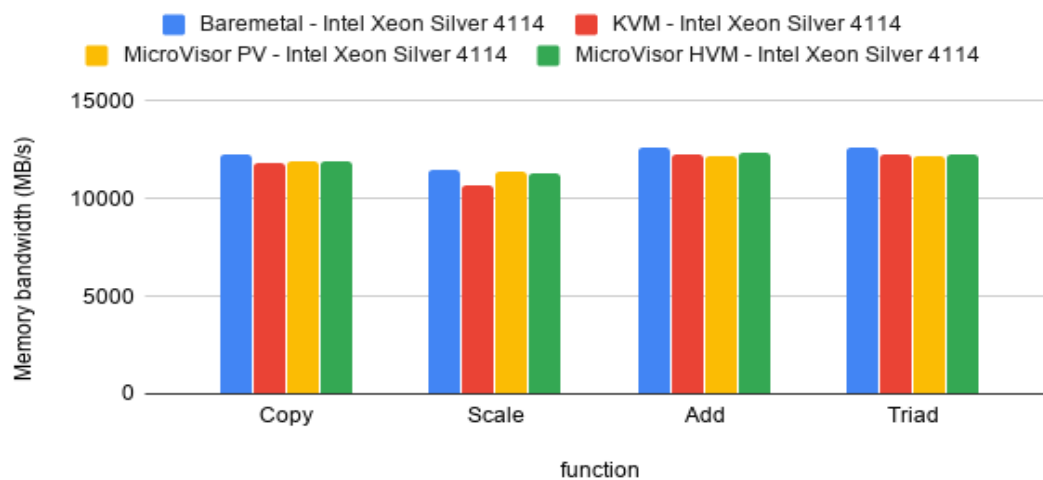


Figure 3.7: STREAM memory bandwidth measurements.

3.2.3 Evaluation of the MicroVisor on the KMAX platform

The goal of porting the MicroVisor directly on top of the KMAX hardware was to combine the underlying hardware capabilities of KMAX with the flexibilities provided by the MicroVisor and to allow the rest of the ACTiCLOUD components (e.g., ACTiManager) and applications to operate efficiently over the KMAX hardware.

As discussed in D4.4: “ACTiCLOUD Final Prototype”, OnApp has worked closely with Kaleao to port all the required components for the MicroVisor platform to the KMAX architecture. This includes the hypervisor, drivers, bootloaders, configuration scripts, OpenStack services with customized drivers, the monitoring statd service, the virxtd agent and the management REST API services.

Systems Measured & Testbeds

In the evaluation of MicroVisor on KMAX we have measured two systems:

- Bare-metal Linux.** This system uses a bare-metal Ubuntu Linux setup on a whole KMAX compute node having an Exynos SoC with 8 CPU cores (ARM64 big.LITTLE architecture) and 4GB of RAM. This compute node hosting this setup is located at the KMAX installation at the University of Manchester. The Linux kernel version used is 4.4.16 (aarch64), as provided by Kaleao for their bare-metal product.

- MicroVisor VM.** This measures performance in a standard Ubuntu 16.04 VM running on the MicroVisor on top of a KMAX compute node (Exynos SoC with 8 CPU cores and 4GB of RAM). The VM has 8 VCPUs, which are mapped on the 8 physical CPUs of the compute node. The VM uses 3.8GB of RAM, the maximum we can allocate for a VM on a compute node, since the hypervisor itself requires some memory for its operation. The Linux kernel version on the VM is 4.4.0-96-generic (standard unmodified kernel of the Ubuntu ARM64 distribution). The compute node hosting this setup is located at the KMAX testbed installed in the OnApp premises in Cambridge, UK.

Both systems were measured using identical KMAX compute node hardware as provided by Kaleao (same CPU cores, RAM, etc.). However, there is an important difference in the storage devices used in each system, which have different characteristics. The bare-metal system in Manchester has been configured to use the Universal Flash Storage (UFS) flash for storage, while the MicroVisor system in Cambridge is using a Solid State Drive (SSD) hosted on the Storage Processing Unit (SPU) of the board (which has slower performance), in order to provide a distributed storage layer. The reason that the Microvisor system is not using the UFS flash storage, is that it requires a custom device driver that is not supported currently by the Microvisor. This difference in storage hardware is the main reason we cannot currently compare directly storage-related metrics between these two systems.

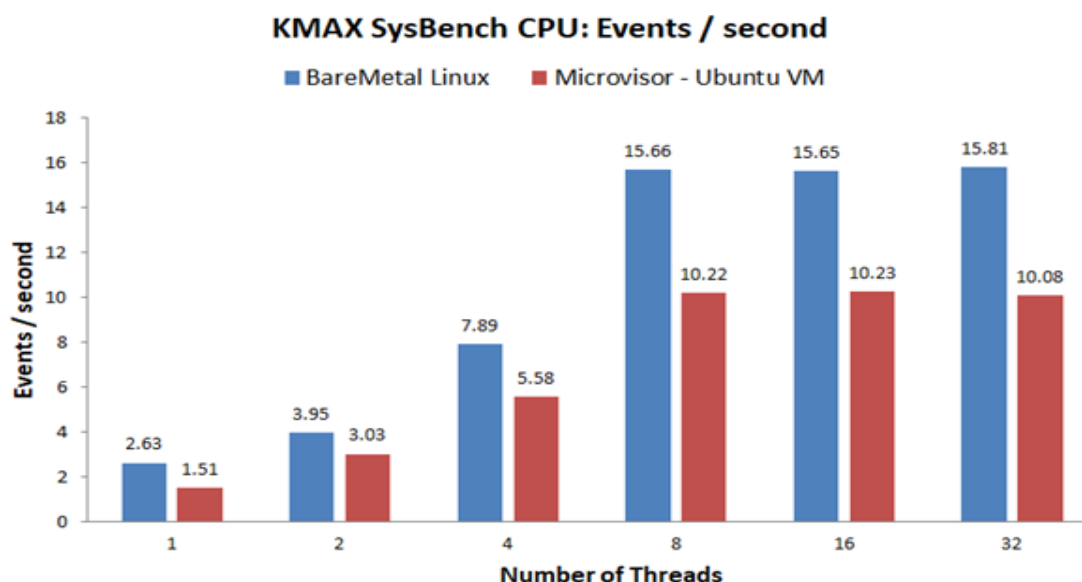


Figure 3.8: KMAX results for SysBench CPU benchmark.

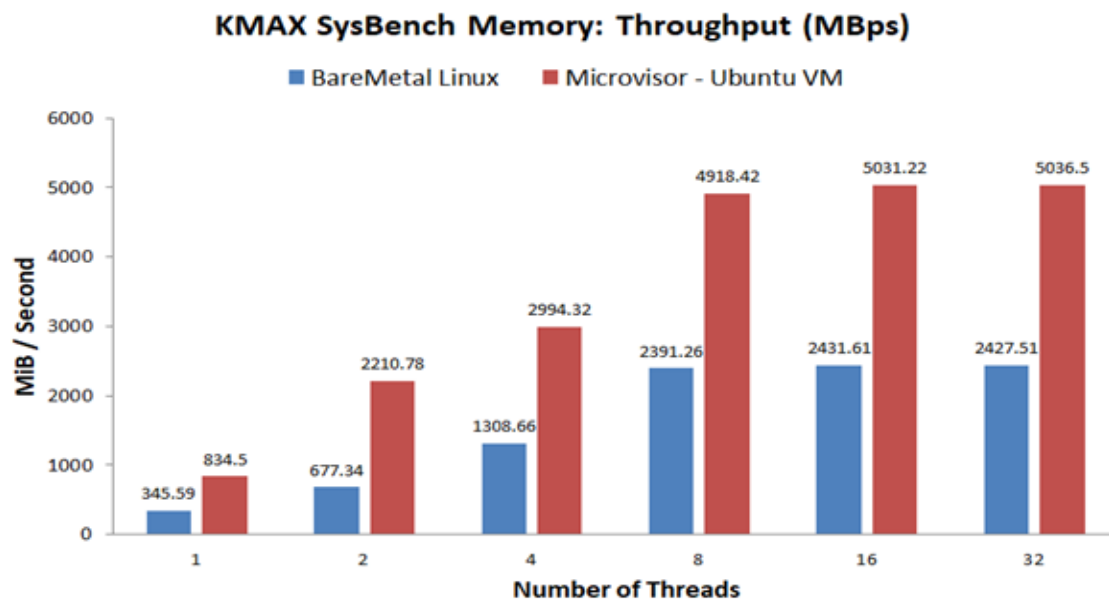


Figure 3.9: KMAX Results for SysBench Memory benchmark

The SysBench CPU and Memory workload results are shown in Figures 3.8 and 3.9 respectively. In the CPU workload we observe that the MicroVisor VM performs worse than the bare-metal Linux, which is reasonable due to the virtualization overhead. We believe that this is due to the VM scheduler not being aware of the big-LITTLE architecture of the system and we plan to improve the performance of the VM in the next update of the MicroVisor through hypervisor and scheduler improvements. In terms of memory throughput we see that the VM outperforms the bare-metal Linux system significantly. Investigating this behavior further, we have seen that the bare-metal Linux installation of Kaleao uses a custom built kernel, with customized CPU and memory bus throttling, customized power management, as well as system libraries built by Kaleao. In the VM case we have used an unmodified Ubuntu distribution, without special CPU and memory throttling, which results in better memory bus performance.

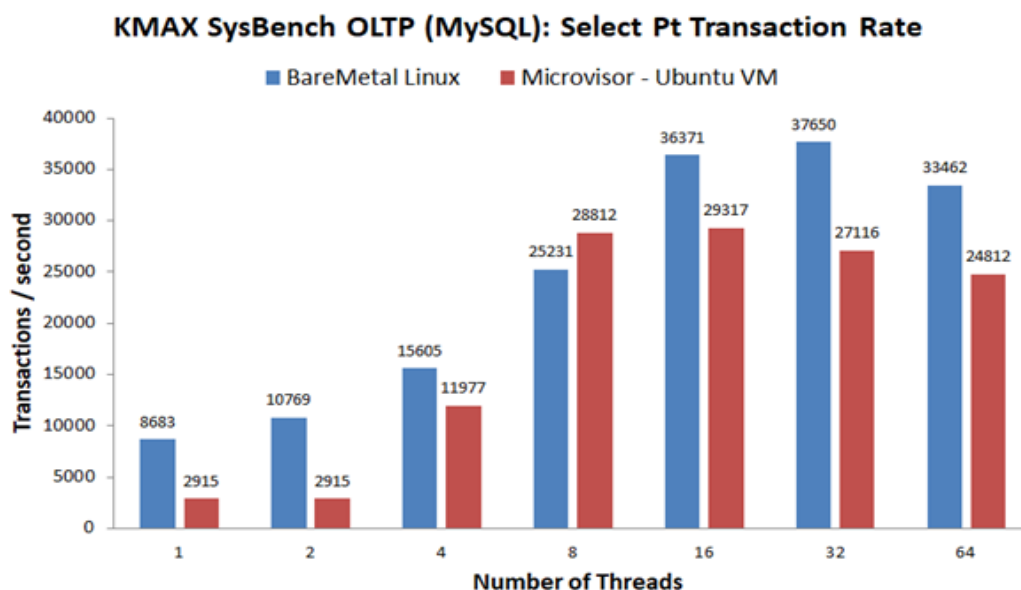


Figure 3.10: KMAX results for Sysbench OLTP Select point benchmark.

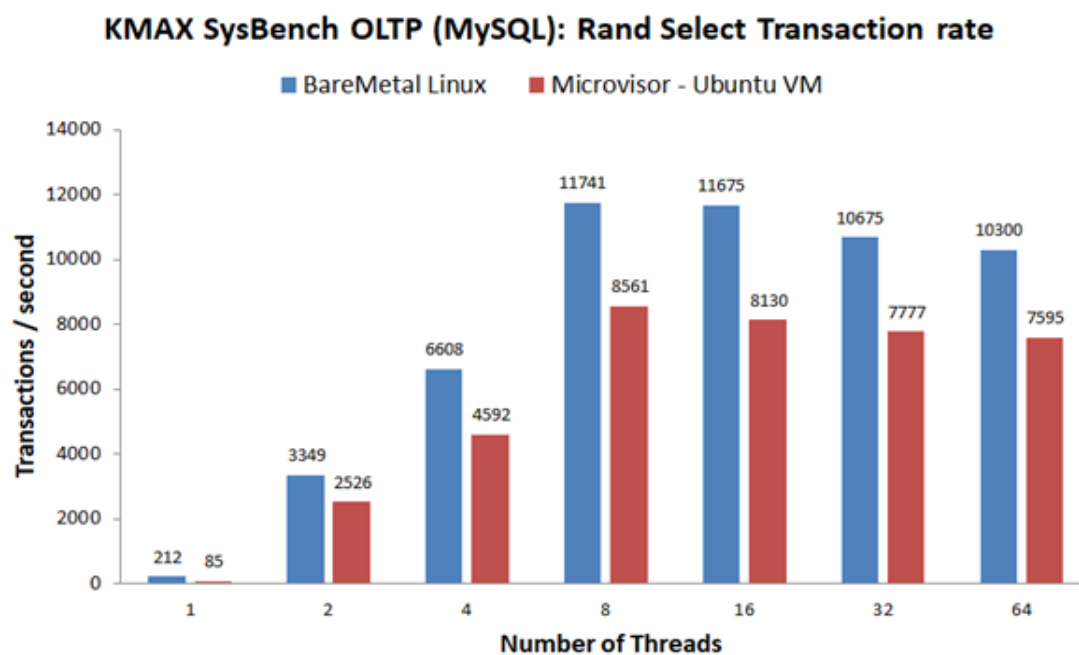


Figure 3.11: KMAX results for SysBench OLTP Random_Range benchmark.

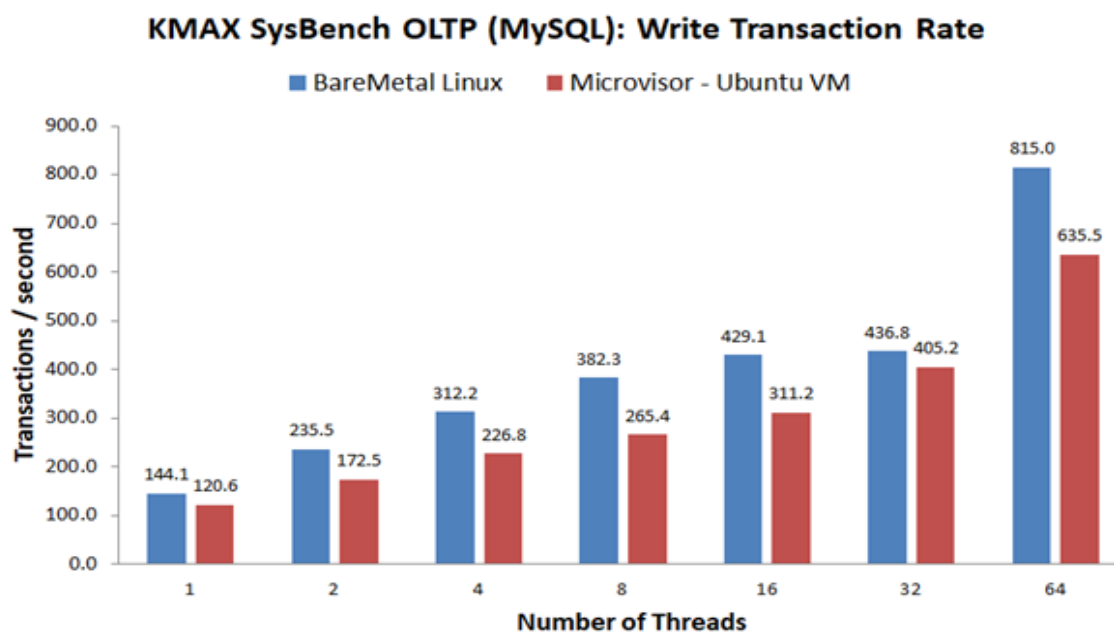


Figure 3.12: KMAX results for SysBench OLTP Write_Only Benchmark.

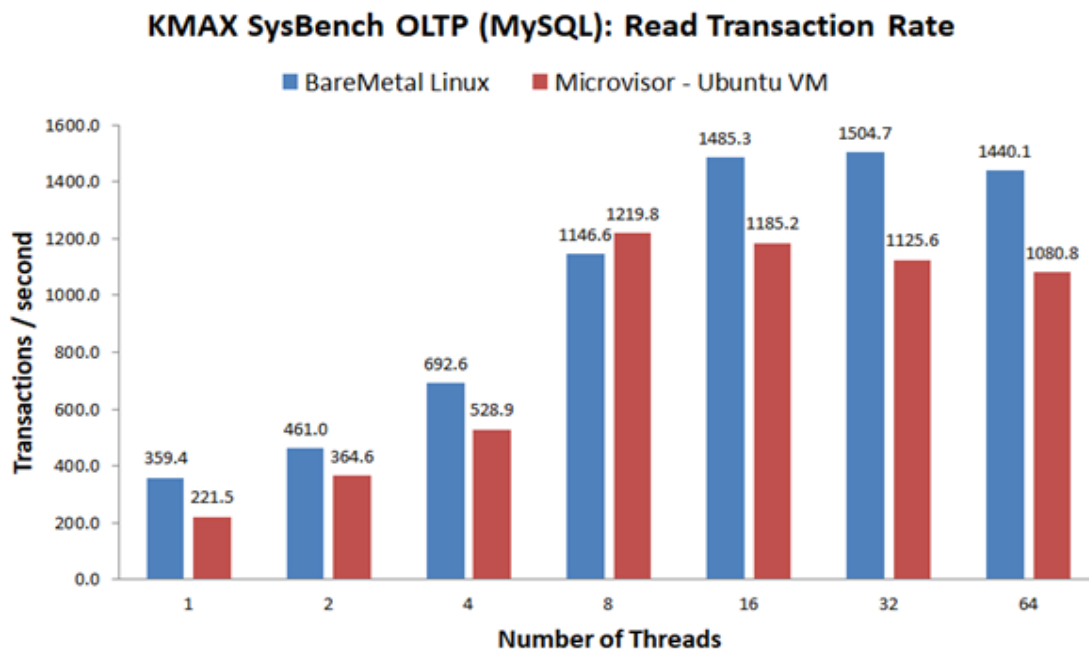


Figure 3.13: KMAX results for SysBench OLTP Read_Only benchmark.

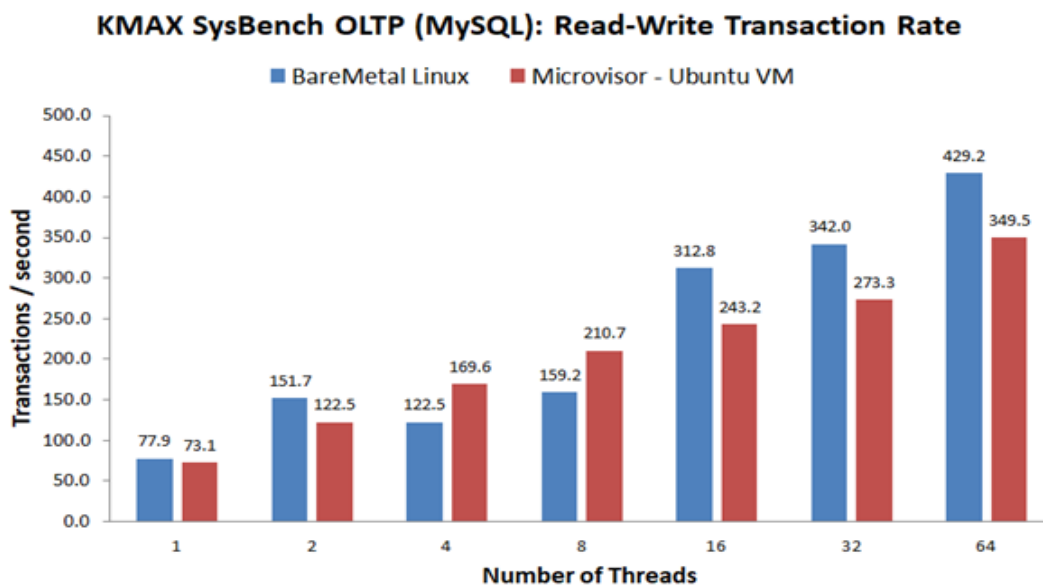


Figure 3.14: KMAX results for SysBench OLTP Read_Write benchmark.

Regarding the SysBench OLTP workload on MySQL, we have used the standard MySQL version provided with Ubuntu (mysql Ver. 14.14 Distrib 5.7.24) and we have some mixed results for the transaction rate (transactions per second), shown in Figures 3.10 - 3.14. In the two Select workloads (mostly read-only) we see that the bare-metal system mostly outperforms the VM, except on the 8 threads run (Figure 3.10), where the VM performs better¹¹. A similar behaviour

¹¹ In addition to the storage performance difference, we attribute this behavior to the process scheduler being big.LITTLE-aware in the bare-metal Linux setup, but big.LITTLE-agnostic in the Microvisor VM case, i.e., which 4 cores are big and which 4 are little. So scheduling threads uniformly on the CPU cores results

can be observed in the other OLTP runs (read-only, write, and read-write), almost always the bare-metal is better, except for some cases (e.g., read-write with 4 and 8 threads in Figure 3.14), where the VM has a higher transaction rate. To explain these results we should also consider the differences in the storage component between the KMAX setups, where the bare-metal case has an advantage. This higher storage performance, the small amount of extra memory (200MB) for bare-metal and the virtualization overhead of the VM, explain why the VM performs worse.

3.3 Accelerated storage access on KMAX

One of the key areas of KMAX innovation during the project was to investigate and implement a mechanism by which storage requests from network clients could be resolved directly on the SSD device without host intervention. In addition to removing the overhead of host managed storage access, this also significantly increased the distributed storage performance capability of KMAX. In Figure 3.15 below we show the performance increase from a CEPH¹² distributed storage solution when the volume is stripped over a single blade of KMAX running CEPH daemons.

Removing the host bottleneck moved the solution bottleneck to the interface between the platforms' NPU and SPU devices (that are described in Deliverable "D1.2: ACTiCLOUD Architecture"). This learning has resulted in a next generation product capable of delivering the full bandwidth capability of the SSD directly to the network interface.

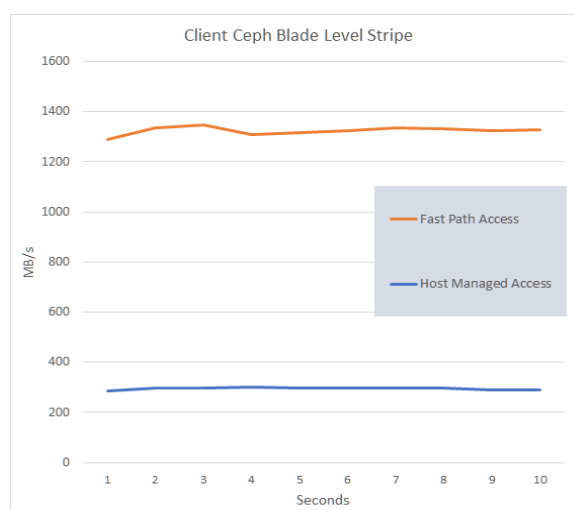


Figure 3.15: Impact on CEPH distributed storage from Fast Path Integration of SSD.

3.4 Resource efficiency with KMAX

The KMAX platform allows the efficient execution of multiple small independent instances of an application, e.g., a MonetDB database, that, although fully capable of operating within the minimal configuration of a cloud instance, there is tension between the required resources, and the cost of the resources that must be powered to deliver the capability. The fine-grained resource units of the KMAX scale-out platform enable the deployment of a higher number of instances, delivering fine-grained energy proportionality with a lower level of resource contention and more predictable performance compared to current cloud implementations. Section 5 provides evaluation results regarding the performance of MonetDB on KMAX.

in non optimal performance when you have less than 8 threads. However, when you have exactly 8 threads the optimal scheduling is using all 8 cores, so the VM performs better.

¹² <https://ceph.io/ceph-storage/>

4 Performance stability

In this section we evaluate how the ACTiCLOUD architecture improves performance stability. We particularly focus on the various components of ACTiManager. We first evaluate the generic version of ACTiManager in improving performance stability. Then we focus on the internal component of ACTiManager targeting the Numascale system, and finally we focus on ACTiManager's feature for offloading/migrating VMs among multiple distributed cloud sites.

4.1 ACTiManager Generic Internal and External Components

4.1.1 Methodology

We use the same methodology that we described in Section 3.1.1 to evaluate how the generic version of ACTiManager improves resource efficiency. Here we focus on evaluating how ACTiManager improves performance stability. We are particularly interested in how much of the executed VMs respected their SLA (user satisfaction). Note that the threshold that we have set for maximum allowed overhead with respect to alone execution for gold and silver VMs is 20% and 400% respectively.

4.1.2 Results

We assess the behavior of each policy under the eight different configurations that were described in Section 3.1.1. Figure 4.1 demonstrates the throughput of each policy on the left axis (workload accepted by the policy) and the average slowdown of each category on the right axis. Within each bar we present the number of VMs that respected their SLAs and the total number of VMs for each class (gold, silver). Note that Figure 4.1 combines Figure 3.2a and 3.2b, and is included here to improve the readability of this section. To make this information more clear, we also include Table 4.1 that shows how many VMs respected their SLAs out of the total number of executed VMs.

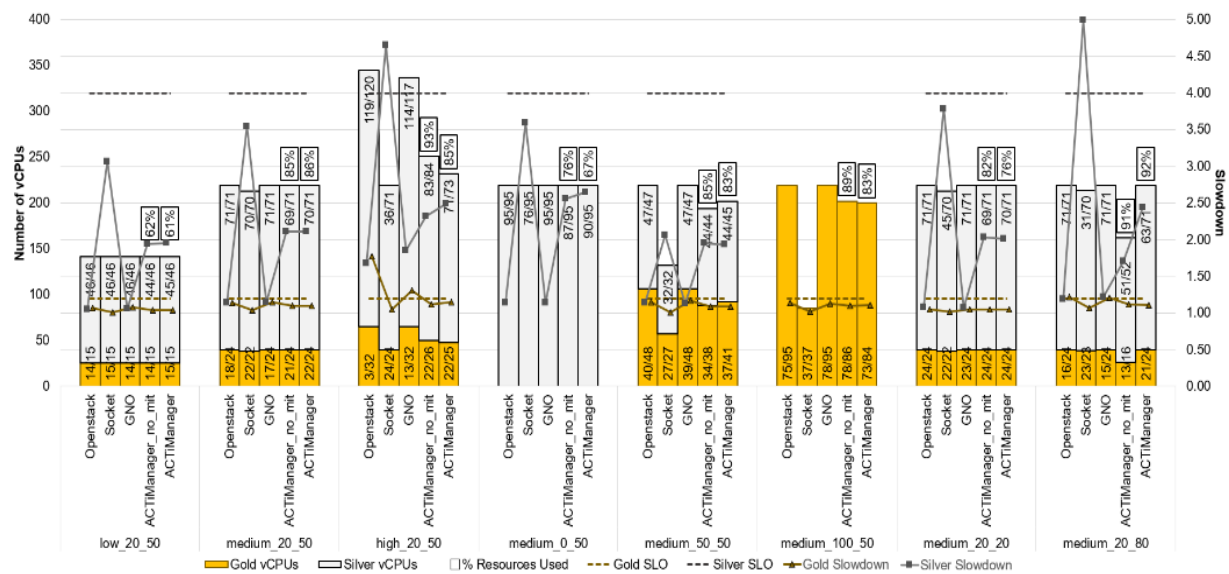


Figure 4.1: (Combines Figures 3.2a and 3.2b) Datacenter throughput and application slowdown. The primary axis (left) shows the total number of all accepted VMs' vCPUs, broken down into gold and silver vCPUs. The secondary axis (right) shows the average slowdown slowdown of applications. Numbers within the bars designate the number of gold/silver VMs that respected their SLAs. The number in the rectangle shows the percentage of the compute resources utilized (shown only for ACTiManager as other policies use 100% of resource).

Table 4.1: Total number of gold/silver VMs that respected their SLAs, out of the total number of gold/silver VMs that were accepted for execution. For example, 14 gold VMs out of a total of 15 gold VMs respected their SLA (up to 20% slowdown compared to standalone execution) with OpenStack for the “low_20_50” execution scenario.

	OpenStack		Socket		GNO		ACTiManager_no_mit		ACTiManager	
	Gold	Silver	Gold	Silver	Gold	Silver	Gold	Silver	Gold	Silver
low_20_50	14/15	46/46	15/15	46/46	14/15	46/46	14/15	44/46	15/15	45/46
medium_20_50	18/24	71/71	22/22	70/70	17/24	71/71	21/24	69/70	22/24	70/71
high_20_50	3/32	119/120	24/24	36/71	13/32	114/117	22/26	83/84	22/25	71/73
medium_0_50	-	95/95	-	76/95	-	95/95	-	87/95	-	90/95
medium_50_50	40/48	47/47	27/27	32/32	39/48	47/47	34/38	44/44	37/41	44/45
medium_100_50	75/95	-	37/37	-	78/95	-	78/86	-	73/84	-
medium_20_20	24/24	71/71	22/22	45/70	23/24	71/71	24/24	69/71	24/24	70/71
medium_20_80	16/24	71/71	23/23	31/70	15/24	71/71	13/16	51/52	21/24	63/71

low_20_50: In this configuration there is limited need for a sophisticated resource allocation policy. ACTiManager preserves performance stability for all gold VMs and for all except than 1 silver VMs, offering performance stability similar to that of the "socket" policy.

medium_20_50: In this scenario that starts to put pressure on the resource managers, we notice that OpenStack and GNO penalize the execution of gold VMs that are suffering from interference, while Socket severely penalizes silver VMs since it largely oversubscribes them, also disregarding any impact from interference. ACTiManager keeps the QoS high for 22 out of 24 gold VMs and for 70 out of 71 silver VMs.

high_20_50: Under high load, the operation of Openstack, Socket, and GNO collapse as they are not able to keep the QoS for the majority of gold VMs, while accepting a large number of VMs. ACTiManager keeps a good balance between throughput, application performance, and resource utilization.

medium_0_50: This scenario does not involve any prioritization of the VMs, as all of them are considered as silver ones. Hence, in this scenario any performance stability improvements come from eliminating performance interference. However, because the VMs are silver, ACTiManager deals with interference in a more relaxed way in order to improve resource efficiency as well. In this scenario all policies operate similarly.

medium_50_50: When increasing the percentage of gold VMs in the mix, Openstack and GNO face problems in keeping their QoS, while Socket becomes more conservative to deal with this load. ACTiManager, again, operates in a more balanced way.

medium_100_50: This scenario does not involve any prioritization of the VMs as all of them are considered as gold ones. Hence, in this scenario any performance stability improvements come from eliminating performance interference, as in the medium_0_50 scenario. However, because the VMs are gold, ACTiManager deals with interference in a stricter way, i.e., it allows less interference to occur than in the medium_0_50 scenario that involved only silver VMs. Indeed,

on this configuration scenario with only gold VMs, Openstack and GNO again face problems and Socket becomes highly conservative, being able, though, to have a perfect QoS for the applications (as expected). ACTiManager approximates the aggressiveness of OpenStack and GNO, keeps the QoS high, but pays some cost in SLA violations.

medium_20_20: In this configuration Openstack, GNO and ACTiManager behave similarly. However, Socket penalizes severely the silver VMs.

medium_20_80: This configuration puts pressure with a more contentious and sensitive workload. OpenStack and GNO penalize the performance of gold VMs. Socket keeps the QoS of gold VMs but fails to preserve the QoS of silver VMs. ACTiManager performs better, with a limited number of SLA violations for gold and silver VMs.

4.1.3 Summary

Overall, ACTiManager achieves performance stability for the majority of the VMs adhering to their respective SLAs.

4.2 ACTiManager.Internal on Numascale

Here we present the performance results for ACTiManager.internal mapping algorithm in a Numascale machine. ACTiManager.Internal treats the mapping as a continuous process where the resources mapped to VMs can be adjusted depending on runtime variations of applications performance and system usage. As the Numascale system behaves as one big machine, the mapping problem consists of mapping vCPUs to physical cores, and if necessary, migrating memory to bring it closer to the cores.

The Numascale system used in the evaluation is a 6 node system with the following setup.

- IBM x3755 M3 servers, each featuring:
 - 3x AMD 6380 (48 cores)
 - 192 GB RAM
 - 1 TB HDD
- 6 Numascale NumaConnect N323 network adapters
- 1 Numascale Management Appliance

The system has 288 cores and 1176 GB of RAM and is connected in a 2-dimensional torus fashion.

To emulate a typical cloud environment as well as exploit the added values of such hardware, we use four VM types: small, medium, large and huge. The configuration for the four VM types is shown in Table 4.2. Note that the huge VM type goes beyond the physical server boundary (i.e., its size is one and half physical servers) to show the capabilities of disaggregated infrastructure to host resource demanding applications (e.g., graph databases) while the rest are designed to emulate VM sizes that are typical in commercial cloud offerings. To load the system, 12 small VMs, 4 medium VMs, 2 large VMs, and 2 huge VMs were hosted at the same time. We consider all VMs to have equal priority, i.e., they are all gold or silver. All guest VMs run the Ubuntu OS.

Table 4.2: VM types used for the experiment.

VM Type	No. cores	Memory (in GB)
Small	4	16
Medium	8	32
Large	16	64
Huge	72	288

For this evaluation we selected a mix of a real-world application, a microservice demo application and synthetic benchmarks. For the real world application, Neo4j¹³, which is a widely used graph database was chosen. Neo4j is implemented in Java and is indicative of a typical application that benefits from running in a large, disaggregated environment due to its demand for both CPU cores and memory. To generate load for the database we used the LDBC SNB¹⁴. To investigate the performance of today's cloud applications we use the Sockshop Microservices Demo¹⁵ together with a load generator that simulates users shopping for socks. Both Neo4j and Sockshop scale well with increased resource allocation. Finally, two synthetic benchmarks were used, SPECjvm2008¹⁶, a benchmark suite for measuring the performance of a Java Runtime Environment (JRE), which contains several real life applications and benchmarks, and Stream¹⁷, which measures memory bandwidth and the corresponding computation rate for simple vector kernels. All workloads were configured to scale according to the VM type they were deployed on.

The analysis presented below shows the results of different experiments using (i) the default scheduling mechanisms in KVM and Linux as baseline, henceforth known as the "Vanilla algorithm", and (ii) the algorithm developed under ACTiCLOUD project, hereafter called "shared-memory-aware algorithm". The shared-memory-aware algorithm has two variants depending on the choice of performance metric, IPC (Instructions per Cycle) or MPI (Cache Misses per Instructions). To differentiate the type of KPI used under the algorithm, we will use SM-IPC and SM-MPI when referring to the shared-memory-aware algorithm using either IPC or MPI as a decision metric, respectively. All experiments were run three times and the results presented in the graphs are the average of the three runs.

Figures 4.2 to 4.4 present relative performance results for the different applications under the vanilla, SM-IPC and SM-MPI algorithms, respectively. The x-axis in the graphs shows the applications, whereas the y-axis shows the performance of each application relative to the three algorithms employed. For all applications except for Neo4j and Sockshop the results presented in the graphs are for the medium VM type. The VM type used for Neo4j is huge while for Sockshop, small is used.

From Figures 4.2 to 4.4, we observe that for all applications when the vanilla algorithm is employed, performance suffers, while the SM-IPC and SM-MPI algorithms provide better and comparable performance for all applications.

The results also indicate that the performance of an application varies significantly across different runs under the vanilla algorithm, while the variation is negligible using both SM-IPC and SM-MPI. For example, the ratio between the standard deviation and average performance of the runs is above 0.4 for all applications using the vanilla algorithm indicating its unpredictable behavior. On the other hand, this ratio remains below 0.04 for both the SM-IPC and SM-MPI algorithms. Moreover, the performance impact varies depending on the type of applications. For example, performance improves by 215x, 33x, 25x, 34x, 5x, 17x, 8x and 105x using SM-IPC and 241x, 37x, 23x, 34x, 5x, 23x, 8x, and 105x using SM-MPI compared to using vanilla for the Derby, fft, Sockshop, Sunflow, mpegaudio, Sor, Neo4j, and Stream applications, respectively.

¹³ . Miller, "Graph database applications and concepts with neo4j," in The Southern Association for Information Systems Conference (SAIS), vol. 2324, 2013.

¹⁴ <http://ldbcouncil.org/developer/snb>

¹⁵ "Sockshop microservices demo," available: <https://microservices-demo.github.io/>

¹⁶ K. Shiv, K. Chow, Y. Wang, and D. Petrochenko, "Specjvm2008 performance characterization," in Computer Performance Evaluation and Benchmarking, D. Kaeli and K. Sachs, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 17–35.

¹⁷ "Stream," . Available: <https://www.cs.virginia.edu/stream/>

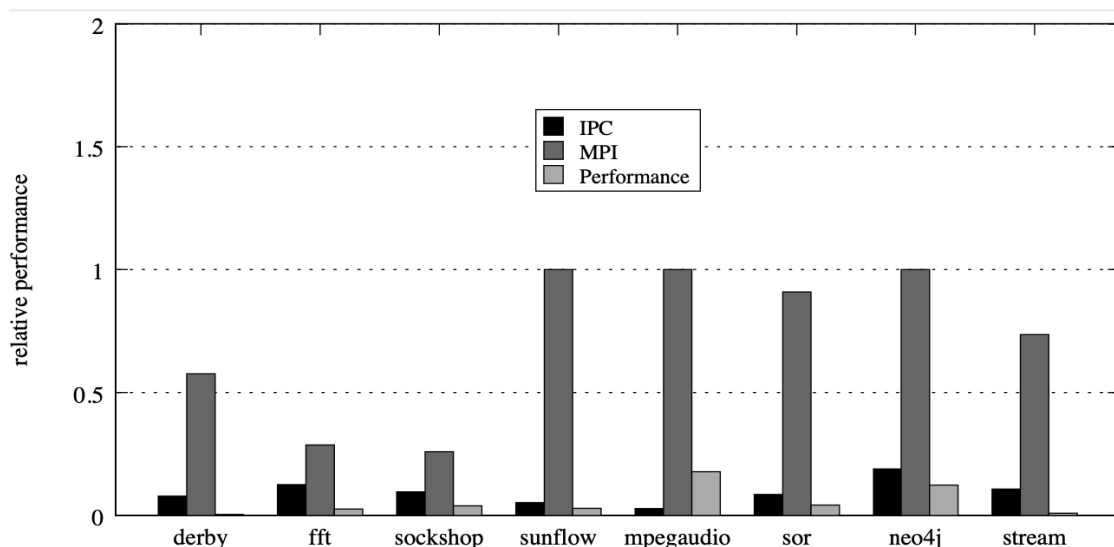


Figure 4.2: Relative performance (IPC, MPI, and application performance) for different applications using vanilla algorithm.

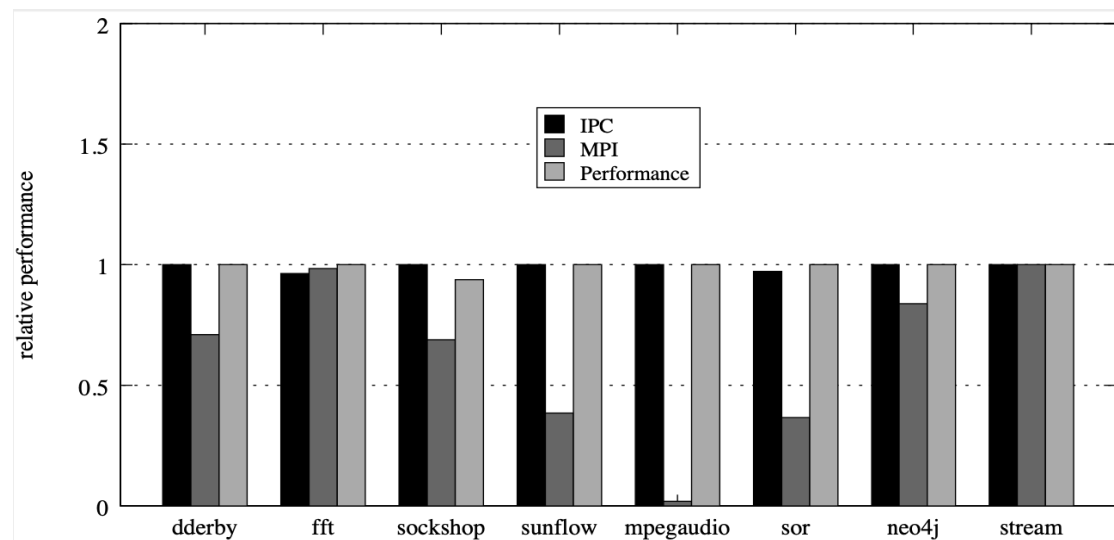


Figure 4.3: Relative performance (IPC, MPI, and application performance) for different applications using SM-MPI.

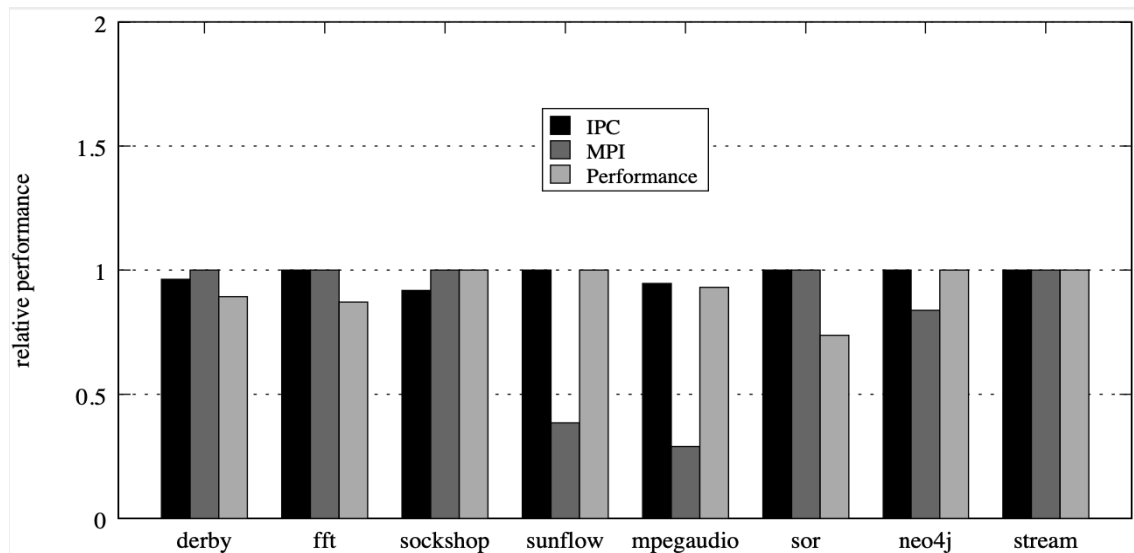


Figure 4.4: Relative performance (IPC, MPI, and application performance) for different applications using SM-IPC.

4.3 ACTiManager.external

ACTiManager.external manages the cloud within a single site and across multiple sites where cross-site offloading is responsible for the latter part. The evaluation in this section shows how ACTiManager mitigates site overload to enforce performance stability. ACTiManager.external attempts to migrate VMs into different nodes within a cloud system. If unsuccessful, ACTiManager.external offloads candidate VMs to a different site.

To perform an experiment that demonstrates this scenario we setup two different cloud sites, cloud-site 1 and cloud-site 2, each with a different OpenStack configuration and ACTiManager.external on top. Cloud-site 1 is slowly overloaded and cloud-site 2 is used as the offloading site. Cloud-site 1 is composed of three physical machines (Nodes A, B, and C), each equipped with a total of 32 cores and 56 GB of memory. In the evaluation scenario we provision two different classes of VMs, gold VMs with 16 vCPU and silver VMs with 1 vCPU. Silver VMs are treated as non-prioritized and therefore these are the VMs that can be migrated and/or offloaded.

The evaluation in this section shows how the ACTiManager.external mitigates overload locally and offloads candidate VMs to a different site when local decision cannot meet policy requirements. In Fig. 4.5, the load average¹⁸ is plotted for all of the nodes in the cluster. The ACTiManager.external is configured with an objective to maintain the overall utilization around 80% which is equivalent to a load average of approximately 26 for cloud-site 1. From the figure, we can observe a load surge from minutes 8 to around 18 on Node-A. Following the load surge, our overload controller tries to perform load-balancing within cloud-site 1 by migrating VMs from Node A to Nodes B and C (reflected between minutes 18 to around 48). This is manifested by the load increase on Nodes B and C and load decrease from Node A during the interval.

The load average of cloud site 1 is still above the threshold so the ACTiManager.external starts offloading to cloud site 2, at around T=40 minutes to T=80 minutes. As cloud site 2 is much smaller and less powerful than cloud site 1, it reaches maximum capacity at around T=70 minutes

¹⁸ Gregg, B.: Linux load averages, <http://www.brendangregg.com/blog/2017-08-08/linux-load-averages.html>

which means that the overall load average for cloud site 1 is still quite high, however it is now stabilized under the threshold. Using an offload site with enough spare capacity that can offset the extra capacity needed by the offloading site to stabilize performance of applications at both sites.

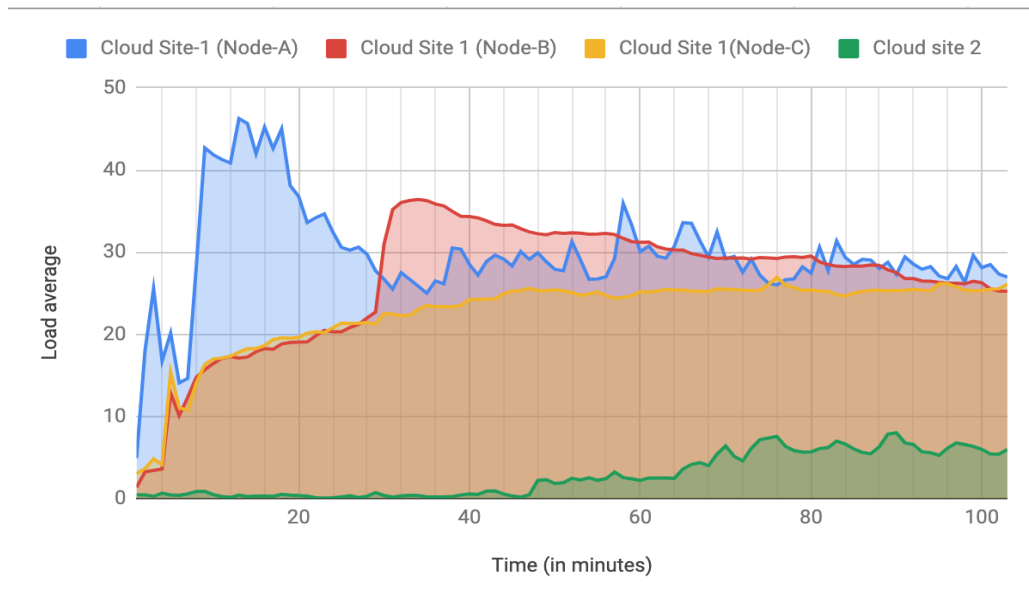


Figure 4.5: In-site load-balancing and cross-site offloading.

5 Scalability in Resource Provisioning

In this section we evaluate how the ACTiCLOUD architecture improves resource scalability. We particularly focus on the following layers of the ACTiCLOUD stack: the system libraries, databases and middleware.

5.1 System Libraries

We evaluate the system libraries developed in ACTiCLOUD by measuring their impact on application scaling using workloads relevant to the ACTiCLOUD scale-up Prototype (Numascale Shared Memory System). This type of architecture is known as scale-up because it aggregates resources from multiple servers, that cannot fit in a single server, and presents them as one server. It is fair to assume that libraries and tests that scale successfully on the Numascale Shared Memory architecture will also improve scaling on HP Superdome Flex and Atos Bullion S16 (that contains Numascale Interconnect). The motive is two-fold:

- To show the scale up effect and relevance of the system libraries in HPC and how this makes running large applications in ACTiCLOUD better.
- To validate that HPC workloads scale well on the ACTiCLOUD scale-up Prototype.

5.1.1 NC-ALLOC

In ACTiCLOUD the major contribution to system libraries has been NC-ALLOC. In order to compare the performance of NC-ALLOC with the standard memory allocator, Numascale has created two synthetic benchmarks:

- **Allocator (AL):** Even if memory allocation often comprises a significant part of an applications' total run-time, when benchmarks are presented (e.g., when running NAS Parallel Benchmark) the allocation/deallocation part of the application is excluded. Numascale wanted to create a microbenchmark that shows this effect. The benchmark stresses allocation and deallocation of threads in parallel. AL uses OpenMP to run on many threads and allocates/deallocates chunks of memory. The allocated memory is committed and returned by the owning thread only. Results are shown in Table 5.1. The core IDs are specified in OpenMP notation. 0-143:1 means that we use cores in the range 0 to 143 with a stride of one (every core), the notation 0-143:2 means that we use the cores in the range 0-143 with a stride of two (every second core), thus we use only 72 out of the 144 cores, and so on. The test in Table 5.1 shows speed-ups, from 2.3x up to 9.3x.
- **Producer-Consumer (PC):** Since it is also interesting to evaluate a more chaotic alloc/free pattern, Numascale created another OpenMP program that separates the threads between "producers" and "consumers", that respectively push to the allocation queue or pop from it. This producer/consumer test helps to monitor how NC-ALLOC improves the efficiency of the communication between the threads. Results are shown in Table 5.2. The results show a small benefit from using NC-ALLOC compared to the system standard libc, but at least no loss of performance in a pure communication setting.

To summarize, NC-ALLOC improves the performance of an application by a varying factor, but very rarely introduces any performance loss.

Table 5.1: allocation/deallocation performance test, al.

# cores	core IDs	Test duration w/ NC-ALLOC (seconds)	Test duration w/ glibc allocator (seconds)	NC-ALLOC speed-up factor
144	0-143:1	4.9	42.0	8.5
72	0-143:2	2.4	18.5	7.7
72	0-71:1	2.8	22.3	7.9
48	0-143:3	5.3	12.6	2.3
48	0-47:1	2.2	20.6	9.3
36	0-143:4	1.8	8.8	4.8
36	0-71:2	1.7	11.6	6.8
36	0-35:1	2.0	15.2	7.6

Table 5.2: NC-ALLOC push/pop performance test, pc.

# cores	core IDs	Test duration with NC-ALLOC (sec.)	Test duration w/ std. mem. allocator (sec.)	NC-ALLOC speed-up factor
144	0-143:1	114	132	1.15
72	0-143:2	58	63	1.08
72	0-71:1	21	23	1.09
48	0-143:3	27	28	1.03
48	0-47:1	8	7	0.88
36	0-143:4	15	16	1.06
36	0-71:2	13	15	1.15
36	0-35:1	4	5	1.25

5.1.2 OpenMP Application

OpenMP is the recommended way of programming a scale-up system. In order to validate the system installation (bootloader, bios, OS and kernel) and hardware platform we show that OpenMP applications can scale well on the ACTiCLOUD scale-up systems delivered by Numascale. In the next section we also add the KVM hypervisor to this validation.

We picked two benchmarks that are well known in the HPC community and are good examples of how to use a scale-up system, the NAS Parallel Benchmarks OpenMP Embarassingly parallel (EP) and STREAM. In Section 5.1.4 we complement these tests by demonstrating scaling of three other workloads from NAS Parallel benchmarks (SP, BT and LU) using the NC-BTL (NumaConnect Byte Transfer Layer) library on the same platform.

NAS Parallel Benchmarks OpenMP

OpenMP is the primary programming paradigm for large SMPs. For most Numascale installations we have created OpenMP patches in order to achieve correct OpenMP placement and improve performance. These patches essentially fix a bug in earlier gcc libgomp implementations (the gcc OpenMP library). The NAS Parallel Benchmarks¹⁹ EP benchmark is used to validate that the OpenMP library (libgomp) works correctly on the ACTiCLOUD platform. EP mimics the

¹⁹ <https://www.nas.nasa.gov/publications/npb.html>

computation and data movement in Computational Fluid Dynamic (CFD) applications. The results presented in Figure 5.1 show that the throughput (Mop/s total) of EP doubles when doubling the number of cores.

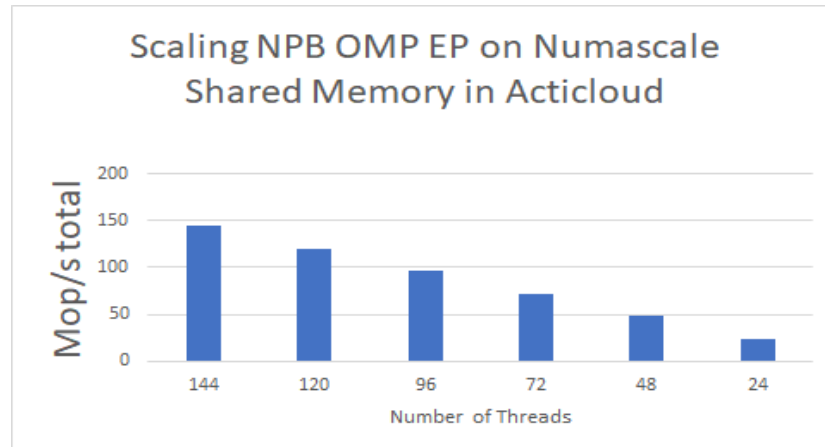


Figure 5.1: NPB OMP EP.

STREAM

The STREAM benchmark is a simple synthetic benchmark that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for four simple vector kernels. Computer CPUs are getting faster much more quickly than computer memory systems. As this progresses, more and more programs will be limited in performance by the memory bandwidth of the system, rather than by the computational performance of the cpu.

As an extreme example, several current high-end machines run simple arithmetic kernels for out-of-cache operands at 4-5% of their rated peak speeds --- that means that they are spending 95-96% of their time idle and waiting for cache misses to be satisfied.

The STREAM benchmark is specifically designed to work with datasets much larger than the available cache on any given system, so that the results are (presumably) more indicative of the performance of very large, vector style applications.

The OpenMP benchmark stream²⁰ scales very well on Scale-up systems like the Numascale Shared Memory system in Athens, the results are given below and show that the memory bandwidth doubles when using twice as many cores.

This shows that the memory bandwidth is high when using system, kernel and hardware implementation for the Numascale Shared Memory Systems in ACTiCLOUD.

²⁰ <https://www.cs.virginia.edu/stream/>

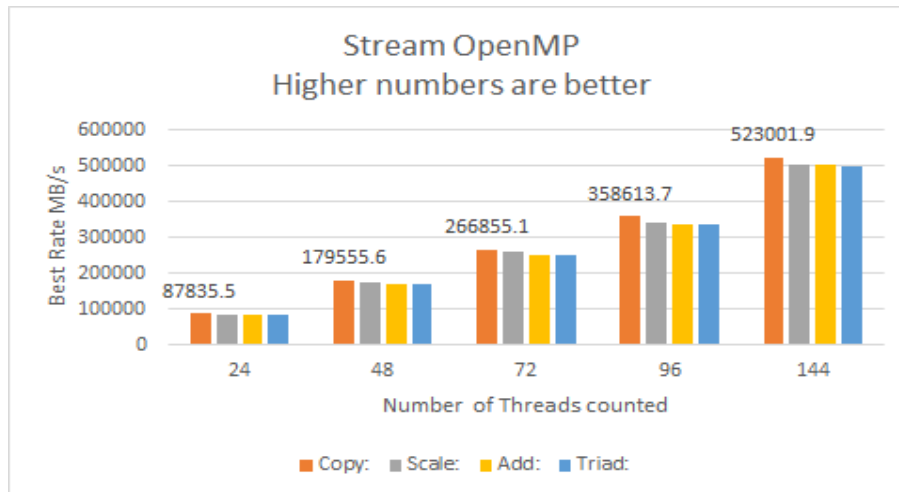


Figure 5.2: Stream benchmark on the Numascale Shared Memory Server in Athens.

5.1.3 KVM

In order to test that a Virtual Machine running KVM still will respect the Numa topology for a VM in a guest OS we can show a map of a VM spanning two Numascale boards (this is equivalent to two Numascale servers). Since the de-facto programming paradigm for these servers is Open MP we have chosen to use the NAS Parallel Benchmark suite in order to demonstrate this. In the first test (shown in the picture below) we specify to use the first 8 cores of the server, hence the `GOMP_CPU_AFFINITY="0-8"` and `OMP_NUM_THREADS=8`. In the top window you can see the application executing, while in the lower window you see that the tool `htop` shows that only the first 8 cores are used.

```

dani... x root... x atle... x atle... x local... x av@... x root... x [scre... x av@... x

NAS Parallel Benchmarks (NPB3.3-OMP) - EP Benchmark

Number of random numbers generated:      8589934592
Number of available threads:              8

EP Benchmark Results:

CPU Time = 45.5987
N = 2^ 32
No. Gaussian Pairs = 3373275903.
Sums = 4.764367927995979D+04 -8.084072988049201D+04
Counts:
 0 1572172634.
 1 1501108549.
 2 281805648.
 3 17761221.
 4 424017.
 5 3821.
 6 13.
 7 0.
 8 0.
 9 0.

EP Benchmark Completed.
Class = C
Size = 8589934592
Iterations = 0
Time in seconds = 45.60
Total threads = 8
Avail threads = 8
Mop/s total = 188.38
Mop/s/thread = 23.55
Operation type = Random numbers generated
Verification = SUCCESSFUL
Version = 3.3.1
Compile date = 16 Dec 2019

Compile options:
F77 = gfortran
FLINK = $(F77)
F_LIB = (none)
F_INC = (none)
F_FLAGS = (none)
FLINKFLAGS = (none)
RAND = randi8

Please send all errors/feedbacks to:

NPB Development Team
npb@nas.nasa.gov

root@acticloud-wg3:~/NPB3.3-OMP# OMP_NUM_THREADS=8 GOMP_CPU_AFFINITY="0-7" bin/ep.C.x

NAS Parallel Benchmarks (NPB3.3-OMP) - EP Benchmark

Number of random numbers generated:      8589934592
Number of available threads:              8

```

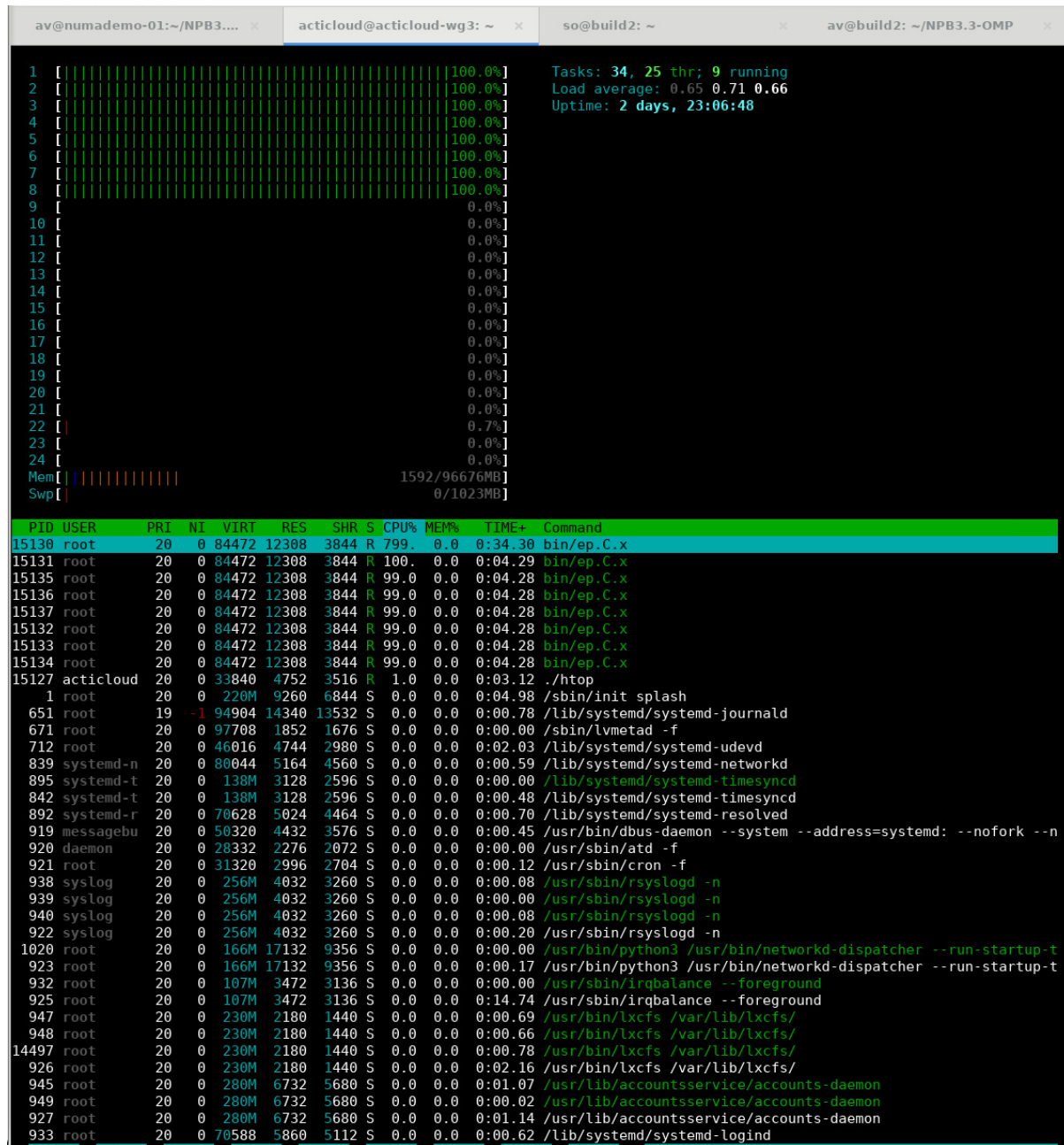


Figure 5.3: Showing that KVM keeps the Numa topology by running two windows running in parallel (OpenMP NPB EP and htop) by showing that every core is used when this is fed to OpenMP through affinity setting.

In the second run shown in Figure 5.4 we specify that we will use every second core by specifying `GOMP_CPU_AFFINITY="0-15:2"` (OMP_NUM_THREADS are still equal to 8). Even inside the VM using KVM we are able to use every second core.

```

dani... x root... x atle... x atle... x local... x av@... x root... x [scre... x av@... x

NAS Parallel Benchmarks (NPB3.3-OMP) - EP Benchmark

Number of random numbers generated:      8589934592
Number of available threads:              8

EP Benchmark Results:

CPU Time = 45.5987
N = 2^ 32
No. Gaussian Pairs = 3373275903.
Sums = 4.764367927995979D+04 -8.084072988049201D+04
Counts:
 0 1572172634.
 1 1501108549.
 2 281805648.
 3 17761221.
 4 424017.
 5 3821.
 6 13.
 7 0.
 8 0.
 9 0.

EP Benchmark Completed.
Class = C
Size = 8589934592
Iterations = 0
Time in seconds = 45.60
Total threads = 8
Avail threads = 8
Mop/s total = 188.38
Mop/s/thread = 23.55
Operation type = Random numbers generated
Verification = SUCCESSFUL
Version = 3.3.1
Compile date = 16 Dec 2019

Compile options:
F77 = gfortran
FLINK = $(F77)
F_LIB = (none)
F_INC = (none)
F_FLAGS = (none)
FLINKFLAGS = (none)
RAND = randi8

Please send all errors/feedbacks to:

NPB Development Team
npb@nas.nasa.gov

root@acticloud-wg3:~/NPB3.3-OMP# OMP_NUM_THREADS=8 GOMP_CPU_AFFINITY="0-7" bin/ep.C.x

NAS Parallel Benchmarks (NPB3.3-OMP) - EP Benchmark

Number of random numbers generated:      8589934592
Number of available threads:              8

```

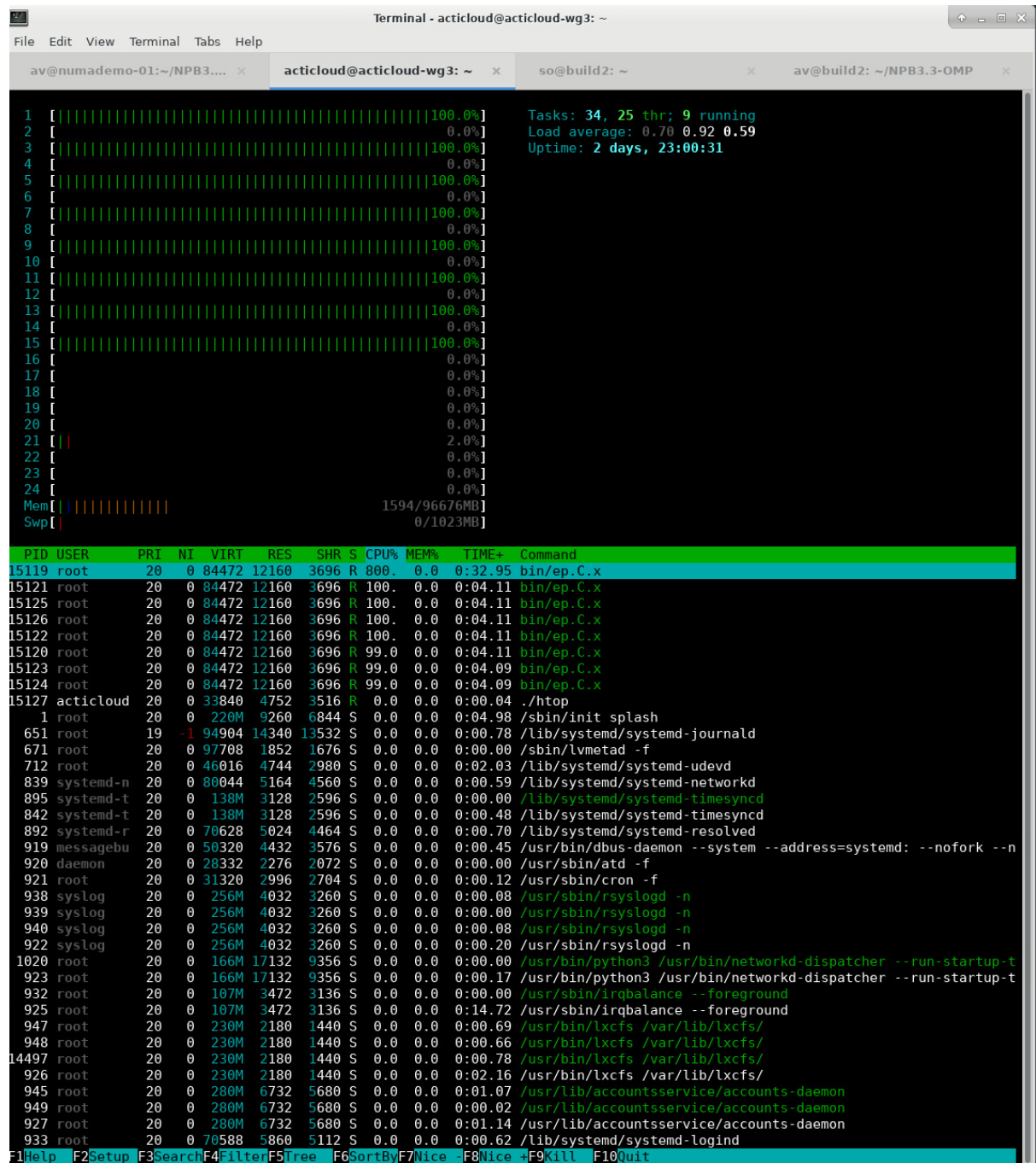



Figure 5.4: Showing that KVM keeps the Numa topology by running two windows running in parallel (OpenMP NPB EP and htop) by showing that every second core is used when this is fed to OpenMP through affinity settings.

Next we wanted to show that we do not lose a significant part of the performance by running inside a VM. We can do this by running the same application as before and comparing it with runs where we are not using KVM, so called “bare metal”. Figure 5.5 shows the results. We observe that the execution inside the VM with KVM shows very little overhead:

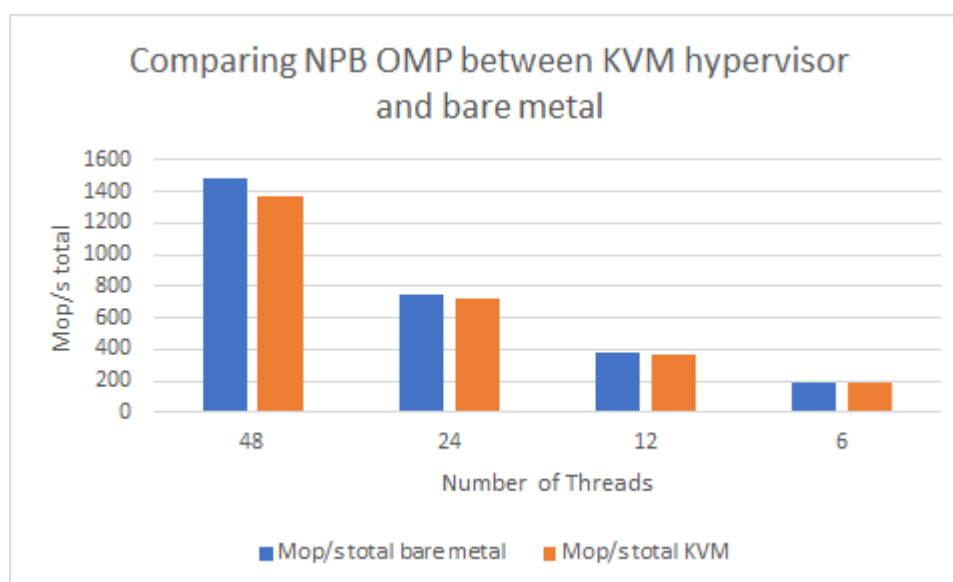


Figure 5.5: Showing that the KVM overhead is negligible on Numascale Shared Memory Installations in ACTiCLOUD.

5.1.4 NumaConnect™ Byte Transfer Layer (NC-BTL)

NC-BTL stands for NumaConnect BTL (Byte Transfer Layer). NC-BTL is used to improve any OpenMPI application that runs with the Numascale fabric: **NumaConnect**. As such we have used the BTL layer in OpenMP to create a module that respects the NUMA topology of Numascale Shared Memory System. It provides a NumaConnect specific BTL module for OpenMPI to exploit features of the hardware architecture to gain the best possible bandwidth and low latency. NC-BTL accelerates significantly OpenMPI.

GROMACS

To show that real HPC applications scale positively on the ACTiCLOUD Scale-up system we have run the popular GROMACS application. GROMACS is a versatile package to perform molecular dynamics, i.e., simulates the Newtonian equations of motion for systems with hundreds to millions of particles. It is an HPC application and scales well with the ACTiCLOUD system library NC-BTL:

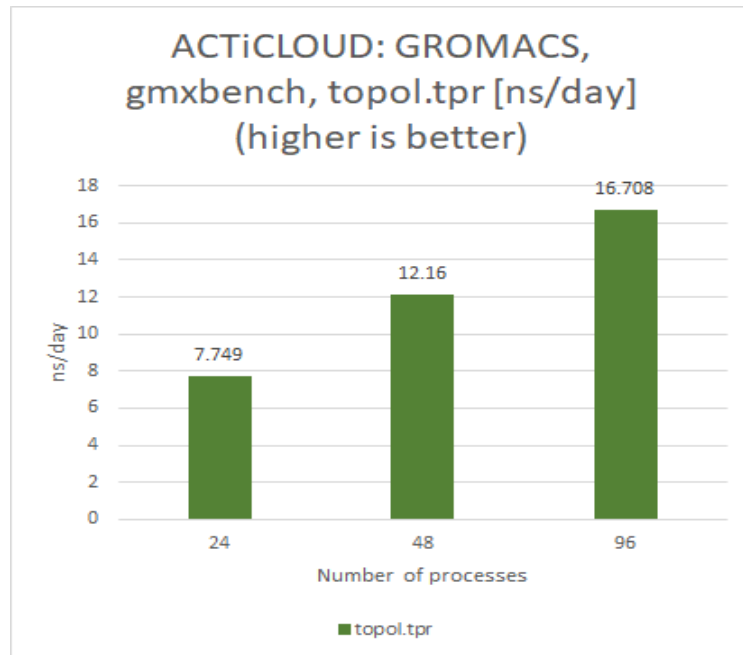


Figure 5.6: GROMACS Scaling on the Numascale Shared Memory system in ACTiCLOUD.

NAS Parallel Benchmarks

In order to further show that MPI applications scale well on the ACTiCLOUD prototype system we have ran some tests from NASA, the NAS Parallel Benchmarks(NPB)²¹ using the ACTiCLOUD system library NC-BTL. We have focused on the three major benchmarks used by NASA in tenders. These are:

- BT - Block Tri-diagonal solver
- SP - Scalar Penta-diagonal solver
- LU - Lower-Upper Gauss-Seidel solver

The system we used for the evaluation has 144 cores divided in six servers (1152 GB RAM):

```
[av@numademo-01 NPB3.3-MPI]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                144
On-line CPU(s) list:   0-143
Thread(s) per core:    1
Core(s) per socket:    8
Socket(s):             18
NUMA node(s):          36
Vendor ID:              AuthenticAMD
CPU family:            21
Model:                 2
Model name:            AMD Opteron(tm) Processor 6380
Stepping:              0
CPU MHz:               2717.901
BogoMIPS:              5000.27
```

²¹ <https://www.nas.nasa.gov/publications/npb.html>

ACTiCLOUD: ACTivating resource efficiency and large databases in the CLOUD

Virtualization: AMD-V
L1d cache: 16K
L1i cache: 64K
L2 cache: 2048K
L3 cache: 6144K
NUMA node0 CPU(s): 0-3
NUMA node1 CPU(s): 4-7
NUMA node2 CPU(s): 8-11
NUMA node3 CPU(s): 12-15
NUMA node4 CPU(s): 16-19
NUMA node5 CPU(s): 20-23
NUMA node6 CPU(s): 24-27
NUMA node7 CPU(s): 28-31
NUMA node8 CPU(s): 32-35
NUMA node9 CPU(s): 36-39
NUMA node10 CPU(s): 40-43
NUMA node11 CPU(s): 44-47
NUMA node12 CPU(s): 48-51
NUMA node13 CPU(s): 52-55
NUMA node14 CPU(s): 56-59
NUMA node15 CPU(s): 60-63
NUMA node16 CPU(s): 64-67
NUMA node17 CPU(s): 68-71
NUMA node18 CPU(s): 72-75
NUMA node19 CPU(s): 76-79
NUMA node20 CPU(s): 80-83
NUMA node21 CPU(s): 84-87
NUMA node22 CPU(s): 88-91
NUMA node23 CPU(s): 92-95
NUMA node24 CPU(s): 96-99
NUMA node25 CPU(s): 100-103
NUMA node26 CPU(s): 104-107
NUMA node27 CPU(s): 108-111
NUMA node28 CPU(s): 112-115
NUMA node29 CPU(s): 116-119
NUMA node30 CPU(s): 120-123
NUMA node31 CPU(s): 124-127
NUMA node32 CPU(s): 128-131
NUMA node33 CPU(s): 132-135
NUMA node34 CPU(s): 136-139
NUMA node35 CPU(s): 140-143

Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm
constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid amd_dcm aperfmperf pni pclmulqdq
monitor ssse3 fma cx16 sse4_1 sse4_2 popcnt aes xsave avx f16c lahf_lm cmp_legacy svm
extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs xop skinit wdt lwp fma4 tce
nodeid_msr tbm topoext perfctr_core perfctr_nb cpb hw_pstate ssbd vmmcall bmi1 arat npt
lbrv svm_lock nrip_save tsc_scale vmcb_clean flushbyasid decodeassists pausefilter
pfthreshold

[av@numademo-01 NPB3.3-MPI]\$

NAS Parallel Benchmark SP

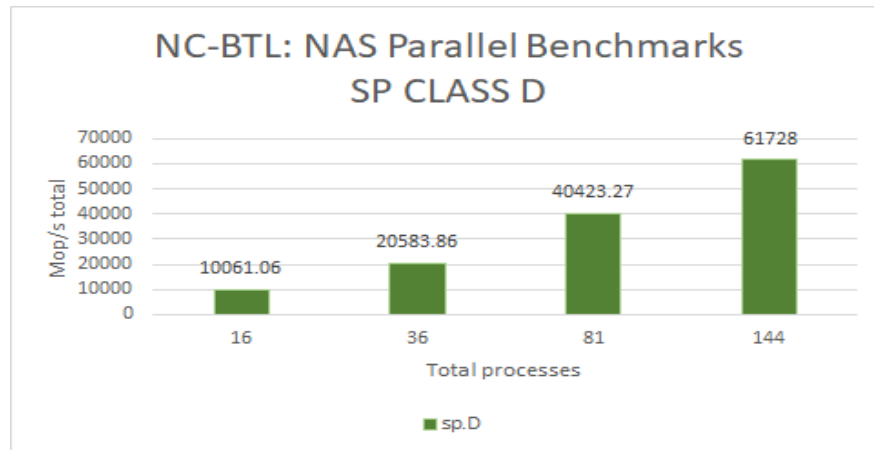


Figure 5.7: NAS Parallel Benchmark SP.

The figure above shows that the HPC Application NAS Parallel Benchmark SP CLASS D scales positively when utilizing more cores on the Numascale system used in ACTiCLOUD when used with NC-BTL. Using all 144 processes we are able to gain 61728 Million operations per second (Mop/s).

NAS Parallel Benchmark BT

The figure below shows that the HPC Application NAS Parallel Benchmark BT CLASS D scales positively when utilizing more cores on the Numascale system used in ACTiCLOUD when used with NC-BTL. Using all 144 processes we are able to gain 147425 Million operations per second (Mop/s).

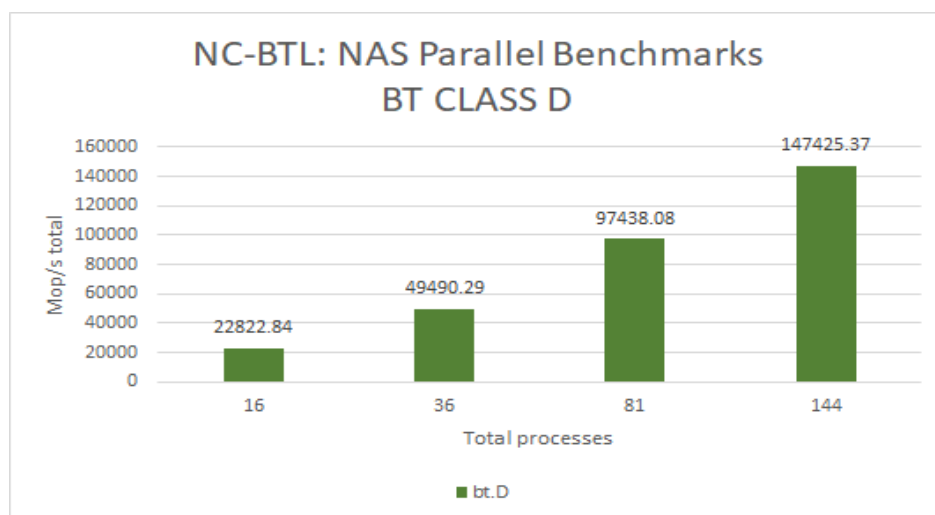


Figure 5.8: NAS Parallel Benchmark BT.

NAS Parallel Benchmark LU

The figure below shows that the HPC Application NAS Parallel Benchmark LU CLASS D scales positively when utilizing more cores on the Numascale system used in ACTiCLOUD when used with NC-BTL. Using all 144 processes we are able to gain 150293 Million operations per second (Mop/s).

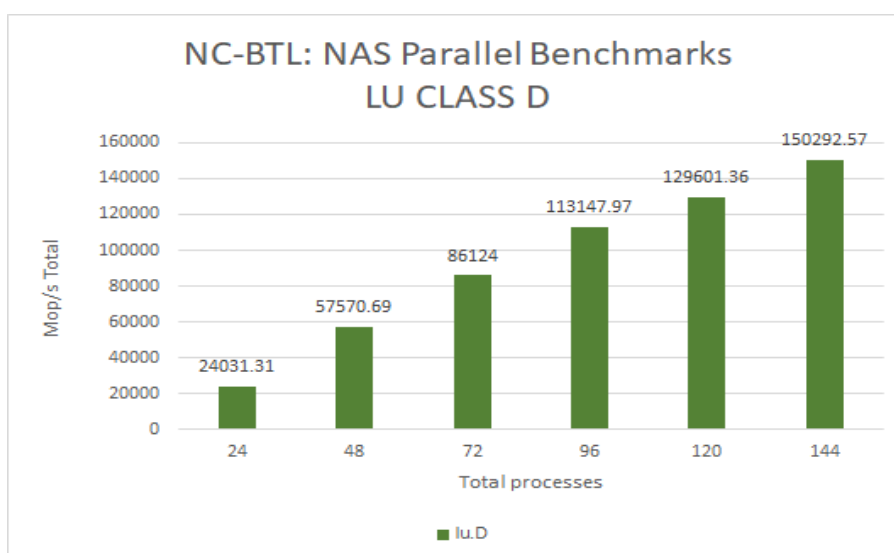


Figure 5.9: NAS Parallel Benchmark LU.

5.1.5 NC-LAPACK

LAPACK²² can solve systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. LAPACK can also handle many associated computations such as matrix factorizations or estimation of condition numbers. This is relevant to popular data analytics applications using the databases in ACTiCLOUD that can store their information in memory instead of inefficient disks, e.g in Monte Carlo simulations.

Similar to LAPACK, ScaLAPACK²³ is a library of linear algebra routines: it solves dense and banded linear systems, least squares problems, eigenvalue problems, and singular value problems. But compared to LAPACK, ScaLAPACK is **designed for parallel distributed memory machines**.

Based **only** on the complex **ScaLAPACK** library for running on parallel distributed memory machines, the **NumaConnect Linear Algebra PACKage** (NC-LAPACK) provides subroutines whose simple calls are similar to those provided by LAPACK, making NC-LAPACK a much simpler library to use than ScaLAPACK.

Table 5.2 is a summary of comparing the ACTiCLOUD system library NC-LAPACK against MKL (Intel Math Kernel Library, the Intel optimized version of LAPACK) on the same calculation. NC-LAPACK demonstrates very good speed-up factors increasing with the number of cores used. NC-LAPACK enables large linear algebra on the ACTiCLOUD system.

Table 5.3: Eigenvalue solver (N=14000) with NC-LAPACK or MKL.

Total #cores used	Repartition of the cores	NC-LAPACK Runtime (minutes)	MKL Runtime (minutes)	NC-LAPACK speed-up factor
144	[0-143:1]	7.9	363.2 (6h)	45.8
48	[0-47:1]	10.3	267 (4.4h)	25.9

²² <http://www.netlib.org/lapack>

²³ <http://www.netlib.org/scalapack/>

5.1.6 Conclusion

The consortium has delivered solid scaling in ACTiCLOUD when using HPC applications and the ACTiCLOUD system libraries. Running these tests using the latest Ubuntu 18.04 or CentOS 8, the KVM Hypervisor does not impose any significant overhead which makes the ACTiCLOUD use of this type of hypervisor a successful choice for scale-up.

5.2 Neo4j on the ACTiCLOUD platform

Neo4j is a graph database, which poses a novel set of challenges when partitioning the data across servers, quite unlike other data models. Since Neo4j is a shared memory database it cannot be evaluated at scale on the KMAX platform. As a result, in this section we focus on the evaluation of Neo4j on the Numascale system using both multiple physical servers as well as a single physical server of the system.

5.2.1 Neo4j on Numascale system with multiple servers

Maintaining good data locality for partitioned graphs is well-known to be a difficult problem in large NUMA machines such as the Numascale server in the ACTiCLOUD platform. During the ACTiCLOUD project we performed numerous experiments to understand the behaviour of Neo4j in order to make it scale up on the ACTiCLOUD platform, attempting to transparently maintain data locality on behalf of the DBMS. In this section we discuss our experiments and present their results, based on which we suggest future work.

The first challenge faced when trying to understand the behaviour of Neo4j on large systems like the ACTiCLOUD platform is finding a representative benchmark suite with a large enough dataset to utilize the platform's resources. Our benchmark of choice for the ACTiCLOUD platform is the LDBC social network benchmark (SNB)²⁴. The Linked Data Benchmark Council (LDBC) is an independent non-profit company that was established as an outcome of the LDBC EU project (Grant Agreement No. 317548)²⁵. Its objective is to establish benchmarks, benchmark practices and benchmark results for graph data management software.

LDBC SNB is a benchmark generating a synthetic social network and performing a set of queries on it. The benchmark includes 14 query templates which it executes with different parameters. The synthetic social network can be configured to be of different sizes. This is controlled through LDBC SNB's scale factor (SF) which ranges from 0.1 to 1000; roughly corresponding to the size of the dataset in gigabytes.

In this scenario we aimed for SF1000 since that results in more than 1 terabyte of data. However, although we were able to generate the dataset while importing it to a Neo4j database disk or file system failures would occur without any indication of the root cause. As a result, we opted for using the next bigger dataset instead which is SF300. SF300 generates about 504GB of data on disk. Although the data is not enough to utilize all the memory of an ACTiCLOUD-enabled Numascale instance, it is sufficient to require more resources than those provided by a single server, allowing it to scale up on the ACTiCLOUD platform.

Taking advantage of the caching support and the large memory of the ACTiCLOUD platform, Neo4j is able to hold in memory the whole dataset and avoid disk accesses. In theory this should allow Neo4j to handle requests more quickly. However, our experimental results do not reflect that. Figure 5.10 plots the results obtained when running the LDBC SNB workload using 150G of

²⁴ <http://ldbncouncil.org/developer/snb>

²⁵ http://cordis.europa.eu/project/rcn/105871_en.html

cache (which fits on a single server) versus using 600G of cache (which needs to utilize memory across multiple servers). To constraint the benchmark on a single server in the first case, and in four neighbouring servers in the second case we use numactl. In our experiment we use query 3 (Q3) of LDBC SNB because it *touches* about 300G of the total 504G of the SF300 dataset, and has the best accessed-data over time ratio. In total we execute 100 iterations of Q3, each with different parameters, from 24 clients running in parallel. The results show that when accessing data across different servers the performance is worse than using a smaller cache and occasionally having to go to the disk.

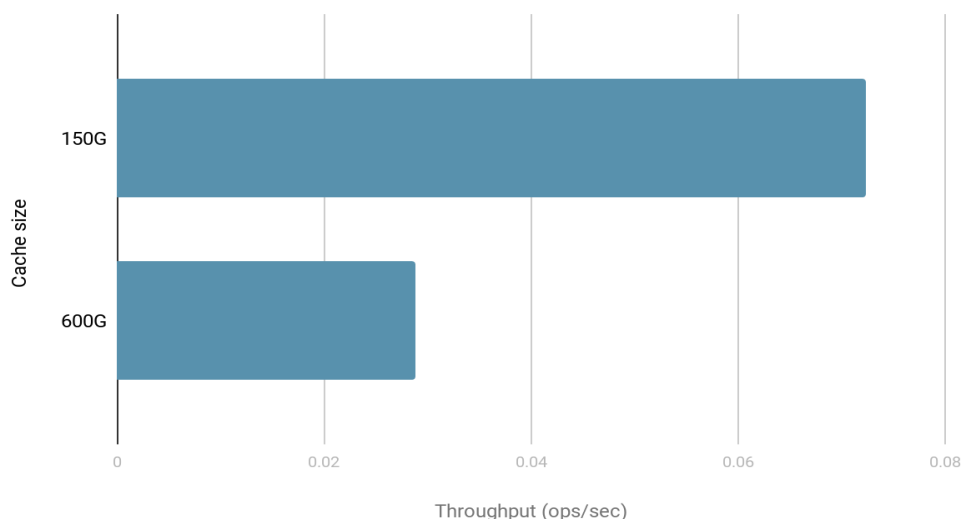


Figure 5.10: LDBC SNB Q3 throughput with cache size 150G and 600G.

Since remote memory accesses (even long distance ones) are faster than disk accesses we developed NumaScope²⁶ to profile and study the experiment. NumaScope allows us to measure the percentage of time that the workload spends waiting on internal resources to become free. The results (shown in Figure 5.11) indicate saturation of the interconnect on the 1st server in our experiments. The Numascale interconnects used in the ACTiCLOUD prototype support up to 16 concurrent remote transactions; after reaching that number new requests get queued. As a result, a significant number of remote memory accesses ends up being queued on the interconnect and taking longer than accessing the disk. Similar behaviour has been observed in other workloads as well.

²⁶ <https://github.com/numascale/numascope>. Numascope is described in ACTiCLOUD deliverable D3.5: ACTiCLOUD-enabled system libraries v2.0

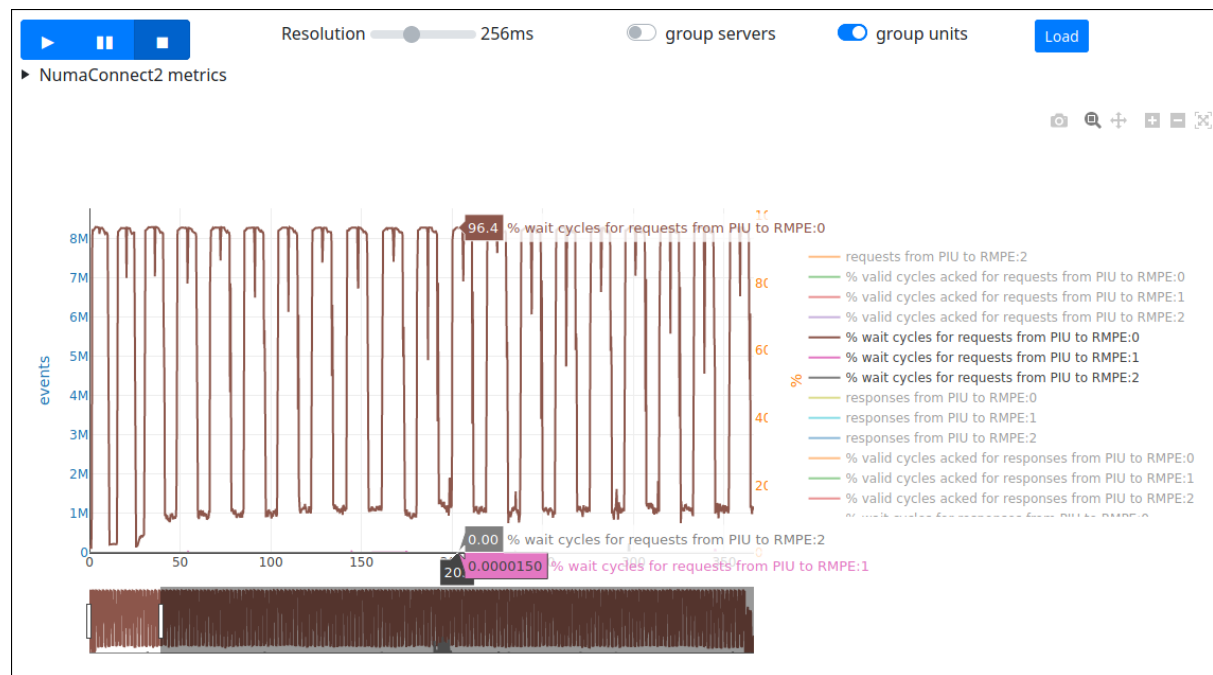


Figure 5.11: NumaScope screenshot.

To avoid similar bottlenecks in the future, based on ACTiCLOUD’s findings Numascale increased the number of supported concurrent remote transactions to 1024 in its next generation interconnects²⁷.

To further understand LDBC SNB’s behaviour on the ACTiCLOUD platform we developed and used the memory profiler of the HyperscaleJVM (see D3.6: *Hyperscale JVM v2.0*). Our study resulted in the following observations:

- Most memory accesses generated by Neo4j concern the cache and are off the Java-heap.
- Even though the Java heap is always able to fit in a single NUMA-node, due to multiple threads accessing it, it ends up being spread across different NUMA-nodes, resulting in remote accesses.

Based on our observations the next steps in optimizing Neo4j on the ACTiCLOUD platform would be to:

- Make the off-heap page cache NUMA-aware, possibly duplicating entries across different servers to reduce remote accesses.
- Use a NUMA-aware garbage collector, like NumaGiC²⁸ to reduce remote accesses.

5.2.2 Neo4j on Numascale system with a single server

In order to demonstrate the utility of the Neo4j-specific work in ACTiCLOUD, versions of the software have been run both in isolation from the ACTiCLOUD stack (both software and hardware) and on top of ACTiCLOUD hardware and software.

In D3.8: *Neo4j ACTiCLOUD extensions v2.0*, early performance benchmarks of Neo4j’s improved Cypher runtimes were presented. These showed several significant performance speedups for the standard LDBC SI benchmark running atop the ACTiCLOUD-enhanced version of Neo4j. In the final benchmarks, almost all of the results are the same within the bounds of experimental error.

²⁷ <https://www.numascale.com/index.php/asic-nc/>

²⁸ <https://dl.acm.org/doi/10.1145/2694344.2694361>

There is a caveat that some read-heavy queries are now twice as fast owing to the removal of a bug from the software. For example on a single Numascale node, we now see better performance:

```
*****

      Numascale

      SF 100 , 1 server (24 CPU)

*****

-----

Group      Benchmark                               Mean  Unit
-----

                BEFORE

-----

ldbc_ish  Fused_Read_2_(runtime,MORSEL)    226.76  s/op
ldbc_ish  Fused_Read_2_(runtime,PARALLEL)   15.34  s/op
-----

                NOW

-----

ldbc_ish  Fused_Read_2_(runtime,MORSEL)    113.49  s/op
ldbc_ish  Fused_Read_2_(runtime,PARALLEL)    7.59  s/op
-----
```

On the Read 2 query above, approximately twice the performance is observed in the final version of the ACTiCLOUD enabled software than the first functionally complete version. While this is a welcome improvement, we do not think it fundamentally changes to directionally good performance improvement that the software provided in earlier test runs, nor do we believe there are other bugs or optimizations remaining to be uncovered that would provide similar discrete performance advantages.

5.3 MonetDB on the ACTiCLOUD platform

The main goal of MonetDB in ACTiCLOUD is to evolve towards a Database-as-a-Service (DBaaS) in the cloud. In such environments, efficiency is even more important than on bare-metal hardware, because it can have a big impact on the actual costs. Furthermore, supporting business analytics applications in the cloud requires the underlying database system to be scalable on both the horizontal and the vertical axis, and can scale gracefully with the available resources.

In the intermediate deliverable D4.3, we have evaluated MonetDB's scalability in both directions on the project hardware systems NUMASCALE and KMAX. In this document, we present some new evaluation results regarding: (i) how the performance of MonetDB evolved in the period of the whole project; (ii) how well cloud environments support MonetDB compared to bare-metal systems; and (iii) how does MonetDB scale down to a micro level. This last experiment in particular is an aspect which has often been overlooked by academic experiments, but is particularly important in real-world business analytics applications where massive swarm-based analytics is a technical frontiere.

5.3.1 From 2017 to 2019 and beyond

During the project, MonetDB extensions developed for ACTiCLOUD (as reported in D3.3 and D3.7, “MonetDB ACTiCLOUD extensions” v1.0 and v2.0) have been gradually added into official MonetDB releases. For instance, the releases in 2018 contain various improvements for distributed query processing (D3.3); the predicate-based data partitioning (D3.7) was released in the Apr2019 version; and the in-memory-only mode and MonetDBLite2.0 (D3.7) are included in the “Default” branch ready for release in H1/2020. However, a new feature should never come at the cost of existing performance. Therefore, we run regression tests to guard each release against performance degradation.

In the previous deliverable D4.3, we have reported results focused on particular use cases, e.g., scaling-out distributed queries. In this last evaluation deliverable, we look back on how the overall MonetDB performance has evolved throughout the ACTiCLOUD project. Table 5.3 shows the execution times of all MonetDB releases since the start of ACTiCLOUD on different TPC-H data sets. The numbers are the total execution time of all 22 TPC-H queries in seconds. The experiments were conducted on a Linux desktop containing an Intel i7 CPU with 4 cores 2.1Ghz, 32GB RAM and 1TB SSD. Table 5.4 shows that since the Jul2017 version, MonetDB has become ~50% - ~115% faster.

Table 5.4: evolution of MonetDB performance during ACTiCLOUD using TPC-H scale factor 1, 10, and 100 on an Intel desktop. The numbers are the total execution time of all TPC-H 22 queries in seconds, hence, smaller is better. “Default” is the current development branch to be released in H1/2020.

MonetDB release	SF-1	SF-10	SF-100
Jul2017	1.38	18.1	490
Mar2018	1.37	18.1	_ ²⁹
Aug2018	1.33	16.0	365
Apr2019	1.04	12.5	310
Nov2019	0.91	11.1	283
Default	1.00	8.7	224

MonetDBLite is the embedded version of MonetDB. MonetDBLite was first introduced in 2016³⁰ to ease the adoption of MonetDB by data scientists. By using MonetDBLite, data scientists can directly control their databases from within their application code; they are no longer burdened by the task of managing a database server. Also, MonetDBLite has a smaller memory footprint than a MonetDB server, and it can be more easily configured to work with different amounts of hardware resources (e.g., limited memory through JVM). MonetDBLite v1.0 (described in D3.3) is a separate library, and it is developed and maintained in a separate repository than the MonetDB server. From v2.0, MonetDBLite and MonetDB have been merged into one library and one repository (described in D3.7), which significantly reduce both the development and maintenance effort.

²⁹ Ran out of memory. MonetDB was killed by the OOM killer.

³⁰ In ACTiCLOUD, we have added MonetDBLite-Java (see D3.3) and ported it to ARM64 architecture, and advanced it to MonetDBLite 2.0 (see D3.7).

Since the target use cases of MonetDBLite are lightweight workloads, we have evaluated the performance of both versions of MonetDBLite using TPC-H benchmark at micro scales, as shown in Table 5.5. The experiments were conducted on a KMAX server, with 4 ARM cores and 4GB RAM. For all but one data set, MonetDBLite 2.0 has become significantly faster than MonetDBLite 1.0. With TPC-H SF1, MonetDBLite 2.0 was having difficulties executing query 20, which took ~11.5 seconds, while MonetDBLite 1.0 took less than 0.2 second. We are investigating this performance degradation; a possible cause is the modified query execution plan generator. As an indication of possible performance gain, we show in the row “SF1*” the total query times without Q20.

Table 5.5: evolution of MonetDBLite performance during ACTiCLOUD using TPC-H scale factor 0, 0.01, 0.03, 0.1, 0.3 and 1 on a KMAX server. The numbers are the total execution time of all TPC-H 22 queries in seconds, hence, smaller is better. “SF1*” is the total execution time of SF1 without the problem query 20, which is under investigation.

TPC-H data sets	MonetDBLite 1.0	MonetDBLite 2.0	Performance gain MonetDBLite 2.0 vs MonetDBLite 1.0
SF0.00	0.12	0.08	50%
SF0.01	1.21	0.21	476%
SF0.03	1.20	0.39	208%
SF0.1	1.77	.073	142%
SF0.3	3.45	1.71	102%
SF1	12.96	16.15	-20%
SF1*	12.81	4.65	175%

5.3.2 MonetDB in the cloud

In the second period of the project, we have integrated MonetDB with OnApp’s MicroVisor platform on the Amazon Web Services (AWS) and have evaluated the integration with different TPC-H workloads. In particular, we ran the terabyte TPC-H workload SF1000 (i.e., 1TB data) on AWS. From these evaluations, we have gained some valuable insight into how a popular public cloud supports large scale analytical workload, and how MonetDB behaves in cloud environments. In this section, we present and discuss the results of the larger scale experiments which were conducted on AWS³¹.

AWS provides on-demand cloud computing platforms and APIs on a metered pay-as-you-go basis. For the experiments reported here, the MicroVisor virtualisation platform was deployed on AWS bare-metal servers on-demand, as a direct replacement for the native AWS hypervisor. In the AWS EC2 environment, the MicroVisor platform supports multi-node clusters. Each node is installed and configured on a full bare-metal server. Multiple nodes are connected, and they communicate with each other using the AWS networking infrastructure. Each node is equipped with a number of network interfaces used for both management and data traffic, for which the MicroVisor platform provides support through specialised NIC drivers. The integration of the MicroVisor-VM-on-AWS platform is described in more detail in D4.4.

³¹ The evaluation of MonetDB-MicroVisor integration on the on-premises hardware was only conducted using small data sets, as a verification of the integration. We omit those results here.

For all experiments, we ran the MonetDB Apr2019-SP1 release on the following bare-metal hardware and VM instances:

- *AWS i3.16xlarge*: the AWS EC2 i3.16xlarge VM instances are provisioned with 64 virtual CPU cores, 488 RAM, 8 x 1,900 NVMe SSD, and costs \$4.992 per hour.
- *AWS i3.4xlarge*: the AWS EC2 i3.4xlarge VM instances are provisioned with 16 virtual CPU cores, 122 RAM, 2 x 1,900 NVMe SSD, and costs \$1.248 per hour.
- *MicroVisor-VM-on-AWS*: uses the MicroVisor hypervisor on an AWS i3.metal server, where a VM running Ubuntu 18.04 was provisioned with 64 virtual CPU cores, 488 RAM, ~3TB virtual disk (2 x 1.5TB disks), and costs \$4.992 per hour.
- *NSCALE-56*: is the project's bare-metal NUMASCALE machine with 1TB RAM, the experiments discussed here only used 56 cores of the 144 CPU cores on this machine, running CentOS Linux.

Figure 5.12 shows some results of running TPC-H SF1000 on a MicroVisor-VM-on-AWS³², an AWS i3.16xlarge VM and an NSCALE bare-metal machine³³. The numbers are the minimal query execution times of 5 runs³⁴. Compared to the AWS i3.16xlarge VM, the results of both the MicroVisor VM and NSCALE are better in terms of execution times. This particular AWS i3.16xlarge instance seems to suffer a lot from performance degradation. Specially, query 10 took ~1.5 hours on average, while it is an ordinary TPC-H query, which normally takes ~30 seconds at this scale. Whether this was a one-time thing or repeating phenomenon, we would not know due to the costs to run experiments at this scale. In total³⁵, the 5 runs took ~11.91 hours on the AWS i3.16xlarge instance and cost ~\$59.43, while on the MicroVisor-VM-on-AWS it only took ~3.84 hours and cost ~\$19.15. Since AWS does not compensate for the costs incurred by bad performance of its instances, this highlights a big advantage of the performance stability that the MicroVisor platform offers.

From Figure 5.12 one can conclude that in cloud environments, performance stability is also a paramount factor for the total costs. This strongly argues for an ACTiCLOUD-enabled cloud platform, on which the ACTiManager would have come into action to avoid the huge spikes in execution times with the AWS i3.16xlarge instance.

Furthermore, since the three systems being evaluated have rather different hardware and software configuration, a better way to compare them is to study their performance for the complete benchmark query set as a whole, instead of focusing too much on the individual queries. Therefore, we show in Table 5.5 the TPC-H performance metric Power@Size³⁶ for the three systems. The Power@size metric reflects a system's TPC-H query processing power (based on a query per hour rate) at the chosen database size, and it is computed as:

³² The detailed evaluation analysis of the MicroVisor on Intel and ARM servers is reported in section 3.2 of this deliverable.

³³ The NSCALE results shown here is only an excerpt of the more extensive evaluation that has been conducted under the lead of the partner Numascale. The complete NSCALE evaluation is reported in section 7 of D4.4 "ACTiCLOUD Final Prototype".

³⁴ We choose to analyse the minimal execution times here, instead of the average values, because they are least affected by the performance fluctuation on AWS.

³⁵ This is only the total run time of the queries. This time does not include the time spent on preparing the database before the queries can be run, which includes, e.g., data generation and data loading time. We choose to not include those times here, because we did not repeat the execution of the preparation steps.

³⁶ http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf

$$\text{TPC-H Power@Size} = \frac{3600 * SF}{\sqrt[24]{\prod_{i=1}^{i=22} QI(i,0) * \prod_{j=1}^{j=2} RI(j,0)}}$$

where $SF = 1000$, $QI(i, 0)$ are the execution time of each query, and $RI(j,0)$ is not applicable in our case. Table 5.6 shows that NSCALE-56 has the best overall performance, followed closely by the MicroVisor platform, which is as expected, since a VM on the MicroVisor has an additional virtualisation cost compared to the bare-metal NSCALE-56. Both NSCALE-56 and MicroVisor-VM-on-AWS are more than 60% faster than AWS i3.16xlarge.

Table 5.6: TPC-H Power@Size performance metric (higher is better).

System	TPC-H Power@Size	Performance gain versus AWS i3.16xlarge
AWS i3.16xlarge	67143.82	-
MicroVisor-VM-on-AWS	108931.16	62%
NSCALE-56	113698.35	69%

TPC-H SF1000 (MonetDB Apr2019-SP1): cloud vs bare-metal

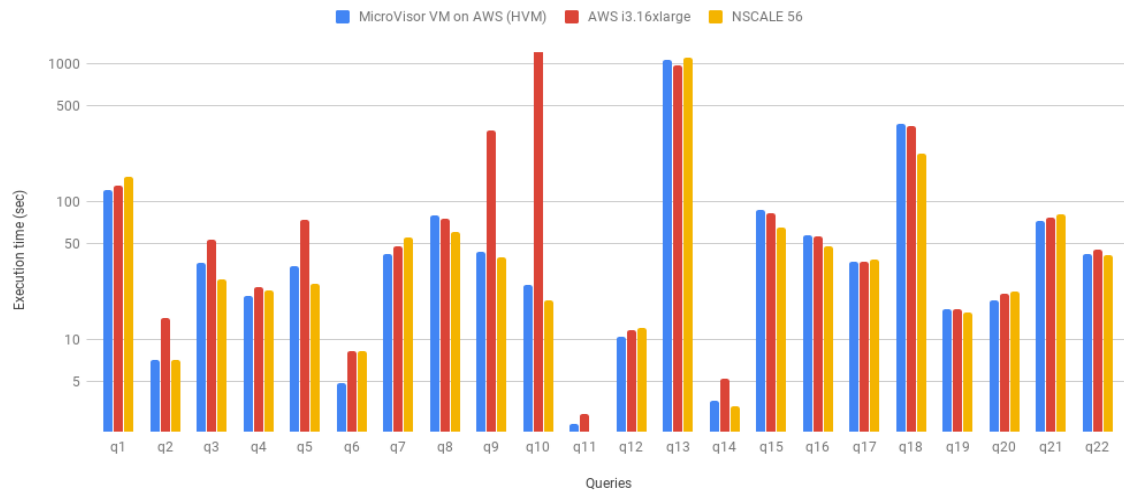


Figure 5.12: TPC-H query execution times in seconds with 1TB data (i.e. SF1000) and MonetDB Apr2019-SP1 release: MicroVisor VM versus AWS EC2 VM versus NSCALE bare-metal.

Queries	Execution time (sec)			Costs		
	SF250	SF1000	Increase	SF250	SF1000	Increase
q1	37,21	130,37	3,50	\$0,0129	\$0,1808	14,01
q2	2,03	14,48	7,12	\$0,0007	\$0,0201	28,49
q3	6,59	52,90	8,02	\$0,0023	\$0,0734	32,09
q4	6,57	24,07	3,66	\$0,0023	\$0,0334	14,65
q5	9,55	74,26	7,78	\$0,0033	\$0,1030	31,11
q6	1,60	8,28	5,18	\$0,0006	\$0,0115	20,71
q7	12,43	47,71	3,84	\$0,0043	\$0,0662	15,36
q8	16,65	75,34	4,52	\$0,0058	\$0,1045	18,10
q9	18,85	329,02	17,46	\$0,0065	\$0,4562	69,83
q10	6,55	4933,63	753,31	\$0,0023	\$6,8413	3013,23
q11	0,73	2,87	3,96	\$0,0003	\$0,0040	15,82
q12	4,44	11,81	2,66	\$0,0015	\$0,0164	10,63
q13	171,11	971,10	5,68	\$0,0593	\$1,3466	22,70
q14	1,23	5,22	4,25	\$0,0004	\$0,0072	16,98
q15	9,41	82,72	8,80	\$0,0033	\$0,1147	35,18
q16	14,06	56,17	4,00	\$0,0049	\$0,0779	15,98
q17	2,92	36,94	12,65	\$0,0010	\$0,0512	50,62
q18	63,25	351,47	5,56	\$0,0219	\$0,4874	22,23
q19	4,96	16,73	3,37	\$0,0017	\$0,0232	13,49
q20	4,18	21,57	5,16	\$0,0014	\$0,0299	20,64
q21	17,10	77,07	4,51	\$0,0059	\$0,1069	18,03
q22	12,51	44,54	3,56	\$0,0043	\$0,0618	14,24

Figure 5.13: SF250 on i3.4xlarge and SF1000 on i3.16xlarge: the execution time and price of each query. The “increase” columns show respectively the increases in time and price from SF250 to SF1000 computed by SF1000/SF250 of the corresponding columns.

Since the virtualised cloud environments enable us to flexibly scale the resources (and hence the price) with the data size and/or workload, we have also conducted a set of experiments with TPC-H SF250 on an AWS i3.4xlarge instance. The query execution times are shown in Figure 5.13 together with the results of SF1000 on i3.16xlarge as comparison. These results reveal some interesting information.

Going from SF250 on i3.4xlarge to SF1000 on i3.16xlarge means we are using 4 times as much hardware to process 4 times as much data. Hence, in theory, one would expect the query execution times to stay more or less the same. However, in reality theory diverges from practice. We have already often seen on bare-metal systems that (TPC-H) query times rarely improve neatly with increasing amounts of hardware. For instance, if it is at all possible to halve query execution times, one often needs (much) more than doubling the number of CPUs or memory. The results in Figure 5.13 show us that this effect is largely intensified in a cloud environment. In this set of experiments, it seems that it is feasible to achieve a time increase of ~4 times³⁷ when we use 4 times as much resources to process 4 times as much data, however, this comes at a cost of ~16 times increase in the price.

The problem here is that, on a cloud platform such as AWS, the price increases linearly with the amount of resources, but the performance does not. This alerts us that in the cloud environments, one should be more conscious when scaling up to bigger data sets. On the other hand, when looking at the results in Figure 5.13 from SF1000 to SF250, it is clear to see how large the gain in both time and cost can be by reducing the data size. This has inspired us to introduce representative database sampling features, so that MonetDB users in the cloud can work with

³⁷ As shown in Figure 5.12, this i3.16xlarge instance has particularly unfortunate performance. The time increases on the comparable MicroVisor VM are closer to 4 times.

much smaller datasets while still getting meaningful results. This is a new project we plan to tackle after ACTiCLOUD.

5.3.3 Scale-down on KMAX

When talking about big data, one would naturally think about all the challenges around efficient processing of large data sets scaling up into terabytes, petabytes and beyond. As a result of this, big data challenges at the micro-scales are an often-overlooked aspect in the big data landscape. In real-world business intelligence applications and data science workloads, big data also frequently comes in the form of a large number of small (or even tiny) data sets. Just as an example, one of our customer databases contains 1.5GB of data spread over more than 600K files, with an average of ~25K per file. At such micro-scales, processing the data itself as fast as possible may no longer be an issue. Instead, the basic overhead of query parsing, compilation and plan generation of a DBMS will gradually become the dominant factor of the total query execution time. Hence, in this set of experiments, our goal is to study the scale-down property of MonetDB, and in comparison with other popular open-source DBMSs.

Naturally, we used KMAX for this set of experiments, since each KMAX server is particularly suitable for lightweight workloads. However, we have consciously chosen to not use KVM (or any other VM technologies) on KMAX, because (i) our goal is to the scale-down properties of different DBMSs, which is orthogonal to the underlying system; and (ii) since the experiments involve very small data sets, the results can be affected by even a small movement in the system. Running the experiments on bare-metal hardware reduces the chance that the results are affected by environmental factors.

		01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22
sf	dbms																						
0.0	monetdb	186.89	125.28	204.70	360.62	191.76	577.23	177.57	132.74	154.41	156.14	nan	179.33	361.73	207.41	231.79	238.42	370.74	241.25	258.77	180.24	181.29	235.67
	monetdblite	206.24	133.46	241.00	412.84	237.18	923.58	213.45	181.81	210.60	175.15	nan	201.63	410.99	283.17	274.50	258.64	440.62	247.10	374.20	235.95	213.49	284.70
	sqlite	8482.20	3739.60	7138.27	9649.34	4628.09	14264.72	3877.34	3189.83	4204.22	5006.88	nan	7885.25	10519.45	12284.86	5802.35	6776.31	13443.75	5956.21	4367.58	5607.06	4548.87	5739.01
	mariadb	3285.70	1825.85	2779.40	3220.10	2335.92	4729.72	1985.49	1841.37	2129.76	2241.53	nan	2841.19	2980.37	3912.81	2103.34	2243.35	3597.15	2321.39	2375.21	2282.20	2122.57	2482.50
	postgresql	1938.48	577.54	1511.33	2070.22	497.83	3032.93	545.68	444.86	393.75	1090.63	nan	1693.05	2184.37	2104.40	1679.17	1638.34	2101.41	904.71	1111.39	1034.41	382.59	1443.90
0.01	monetdb	75.58	89.07	133.41	192.29	111.00	275.37	84.66	101.13	68.01	94.89	187.07	127.30	71.28	149.03	196.05	109.15	203.34	121.03	115.90	109.20	98.01	123.24
	monetdblite	73.82	90.85	141.28	170.39	122.46	200.41	60.81	118.04	85.54	94.95	228.82	71.71	69.27	nan	211.88	127.66	192.46	123.19	76.84	120.78	43.86	148.01
	sqlite	4.86	1236.38	25.11	249.67	238.93	36.49	14.60	20.35	8.53	30.73	71.34	16.89	14.92	36.44	36.68	156.36	17.12	22.87	20.05	0.92	8.75	4.74
	mariadb	3.02	118.31	32.43	11.81	56.86	15.35	71.13	38.54	13.04	13.10	335.63	7.93	16.99	5.15	7.68	102.60	563.82	10.27	254.71	215.21	4.71	177.07
	postgresql	9.67	102.02	40.48	20.82	66.05	35.07	55.96	48.02	16.80	23.76	149.63	20.36	38.89	37.25	36.05	85.08	775.14	10.33	25.29	0.63	26.36	87.33
0.03	monetdb	39.74	81.19	97.84	118.97	71.81	156.24	46.60	78.91	47.77	54.94	179.67	95.62	29.13	116.35	155.23	73.78	186.81	79.68	87.15	84.36	52.47	82.19
	monetdblite	33.35	83.25	93.45	89.31	73.59	98.60	27.12	76.46	51.10	52.55	204.94	37.66	27.28	nan	145.72	84.10	166.52	66.90	38.94	71.69	17.43	91.86
	sqlite	1.56	411.18	8.18	62.29	60.58	12.21	4.57	4.12	1.20	10.15	17.97	5.36	3.47	12.16	12.18	50.30	0.18	7.64	6.37	0.11	2.76	0.39
	mariadb	0.98	31.69	11.77	3.85	18.81	5.27	19.76	12.22	4.18	4.28	108.17	2.71	5.38	1.61	2.59	35.83	168.05	3.38	84.57	76.70	1.56	61.25
	postgresql	4.96	35.90	14.36	6.53	25.47	24.66	18.21	22.33	5.87	13.88	41.55	18.08	12.60	22.84	24.58	29.70	0.23	3.57	17.57	0.07	16.76	30.95
0.1	monetdb	15.84	66.93	54.60	73.33	42.81	82.42	34.05	42.70	21.55	21.13	144.36	49.28	10.28	63.85	76.76	34.19	54.89	22.93	44.14	53.52	23.92	38.30
	monetdblite	14.00	67.39	56.25	60.00	44.24	63.21	24.78	37.95	22.72	21.93	164.78	26.75	10.03	nan	79.88	35.65	45.89	36.81	30.73	19.36	13.80	41.18
	sqlite	0.46	82.89	2.77	18.58	18.41	3.71	0.99	0.43	0.29	3.03	5.24	1.59	0.54	3.66	3.70	13.45	0.01	2.29	1.83	0.01	0.82	0.03
	mariadb	0.29	10.14	3.02	1.12	5.06	1.55	5.20	3.63	0.54	1.20	32.41	0.79	1.56	0.32	0.77	10.54	50.53	0.89	28.48	19.90	0.46	18.86
	postgresql	1.54	12.57	6.87	3.29	10.33	8.74	7.75	7.78	2.74	5.02	12.66	4.10	3.40	7.94	8.54	8.08	0.01	1.20	6.18	0.01	5.27	7.95
0.3	monetdb	5.66	46.14	31.67	36.90	22.91	42.53	16.67	23.36	10.05	9.50	78.13	29.45	8.15	39.38	44.84	12.36	28.49	10.15	27.85	32.42	11.88	18.34
	monetdblite	5.42	44.79	28.78	29.61	21.95	28.81	10.98	17.86	9.60	9.50	74.77	12.77	7.53	nan	40.96	12.97	21.18	13.78	14.34	3.73	6.23	18.67
	sqlite	0.14	27.55	0.91	6.14	6.12	1.26	0.26	0.20	0.08	1.01	1.64	0.53	0.13	1.22	1.23	4.23	0.00	0.76	0.60	0.00	0.27	0.00
	mariadb	0.10	3.69	0.27	0.33	0.67	0.45	0.51	0.32	0.07	0.23	10.86	0.19	0.48	0.01	0.18	3.58	17.04	0.16	6.41	1.12	0.12	6.33
	postgresql	0.51	3.93	3.37	6.65	5.30	2.92	3.67	3.92	1.24	1.76	7.96	1.96	0.99	2.93	2.73	2.55	0.00	0.43	2.24	0.00	1.83	3.48
1.0	monetdb	1.81	27.28	13.51	16.09	7.81	18.88	5.61	10.15	2.48	3.32	43.91	13.04	1.66	19.44	19.96	5.01	10.01	1.31	nan	nan	nan	6.65
	monetdblite	1.79	25.79	11.61	11.87	7.18	10.93	3.48	6.84	2.44	3.38	41.41	4.47	1.50	nan	17.58	5.01	6.21	2.92	4.56	0.09	2.13	6.41
	sqlite	0.04	8.10	0.28	1.84	1.84	0.36	0.07	0.01	0.02	0.30	0.44	0.16	0.03	0.36	0.36	1.24	0.00	0.23	0.18	nan	0.07	nan
	mariadb	0.03	0.12	0.04	0.21	0.06	0.14	0.11	0.04	0.01	0.01	0.91	0.06	0.02	0.00	0.07	0.57	2.18	0.02	0.78	0.22	0.04	1.87
	postgresql	0.15	1.03	0.72	1.93	1.56	0.81	0.77	0.88	0.31	0.51	2.47	0.54	0.57	0.85	0.76	1.08	nan	0.10	0.65	nan	nan	1.03

Figure 5.14: TPC-H scale factors 1.0, 0.3, 0.1, 0.03, 0.01, and 0.0 (i.e. 1GB, 300MB, 100MB, 30 MB, 10MB and 0MB) with MonetDB, MonetDBLite, SQLite, MariaDB, PostgreSQL on one KMAX server with 4 ARM cores and 4GB RAM. The numbers shown are queries-per-sec for each query, hence, larger is better.

Figure 5.14 shows all results of this set of experiments, which was run on one KMAX server with 4 ARM cores and 4 GB RAM. The left-most two columns show the scale factors and the DBMS. We have run TPC-H scale factor 1.0, 0.3, 0.1, 0.03, 0.01 down to 0.0 (i.e. 1GB, 300MB, 100MB, 30MB, 10MB and 0MB), and used MonetDB (Nov2019 release), MonetDBLite2.0, SQLite 3.28.0, MariaDB 10.0 and PostgreSQL 12.1. In the remaining part of this table, we show the ops-per-sec³⁸ of each TPC-H query for each SF + DBMS combination. Ops-per-sec is a standard measurement of the Java Microbenchmark Harness (JMH) framework³⁹ that we have used for these experiments.

In Figure 5.14, the “nan”-s of q11 at SF 0.0 (i.e. an empty database) are because q11 contains a division by the scale factor, while the “nan”-s at SF1.0 are caused by time out after >1 hour. Nevertheless, these results show some clear trends.

First, at the smallest scale (i.e. SF0.0), SQLite is by far the fastest system, which is more or less expected, since SQLite is the most popular embedded DBMS on mobile devices. Although PostgreSQL and MariaDB are much slower than SQLite, they are still an order of magnitude faster than both MonetDB versions. Their performance would be our target when we try to reduce MonetDB’s basic overhead. Second, when the data size grows, the highest ops-per-sec quickly shifts to MonetDB. This is a trend we like to see. Finally, except at SF0.0, MonetDB often has a higher ops-per-sec than MonetDBLite, which is unexpected. The small differences can be explained by the fact that the scales of these experiences are so small that their results are susceptible to even some small performance fluctuations in the system. However, the larger differences, e.g. those of q6 and q7, are worth future investigation.

5.3.4 Summary

To summarise, in this section, we have evaluated the evolution of MonetDB’s performance during the project with ACTiCLOUD extensions; studied the behaviour of analytical workloads on different cloud platforms; and examined MonetDB’s behaviour with workload scales and system resources which are far outside MonetDB’s comfort zone. Instead of drawing simple conclusions like system X is better than system Y, the experiment results presented in this section have helped us to gain insight in new environments (i.e. cloud platforms) and a broader scope of analytical workloads (i.e. also at the micro level). In conclusion, thanks to the work we have done both within MonetDB code bases, as well as together with project’s hardware and virtualisation partners, MonetDB has taken her first steps towards a DBaaS in the cloud.

5.4 Hyperscale JVM

Hyperscale JVM acts as the middleware for JVM-based applications running on the ACTiCLOUD architecture. As a result its performance is critical and directly affects the performance and scalability of the applications running on top of it. In this section we present the evaluation of the Hyperscale JVM on the ACTiCLOUD platform.

The purpose of this evaluation is to demonstrate the maturity of Hyperscale JVM, through the high pass rates for the different benchmarks, as well as to give an estimation of its performance compared to the competition. Additionally, we evaluate the Hyperscale JVM memory profiler, which can be used to understand the memory behavior of Java applications and modify them to improve data locality in order to achieve better scaling on the ACTiCLOUD platform, as well as on other NUMA-aware systems.

³⁸ The operation here is a query, hence, ops-per-second equals queries-per-second in this case.

³⁹ <http://openjdk.java.net/projects/code-tools/jmh/>

5.4.1 Evaluation platforms

We evaluate Hyperscale JVM on both the Numascale and the KMAX platforms. For the evaluation on the Numascale platform we deploy our benchmarks on the Oslo testbed (described in Section 4.2), while for the evaluation on the KMAX platform we deploy our benchmarks on the Manchester testbed (described in Section 3.2.3).

5.4.2 Evaluation methodology

Since Hyperscale JVM is based on the state-of-the-art research JVM, MaxineVM⁴⁰, we evaluate against both the production-quality HotSpot JVM and the competing research JVM, JikesRVM⁴¹. For the evaluation we use the DaCapo⁴² benchmark suite (for single server experiments), as well as the LDBC SNB⁴³ running against Neo4j (for large scale experiments). When using the DaCapo benchmark suite we use the command line parameters `-C` and `-t 4`. The first one instructs the benchmark harness to run enough times until the VMs become warm and the measurements are stable. The second one instructs the harness to use 4 cores, i.e. the number of cores available on a single numa-node on the Numascale testbed. For LDBC SNB we use the same setup described in Section 5.2.1.

5.4.3 Performance evaluation against HotSpot, the standard production-quality JVM

To evaluate the performance of HyperscaleJVM we compare its performance against the production-quality Hotspot JVM.

Oslo testbed

DaCapo-9.12-MR1-bach

Hyperscale JVM on the Oslo testbed successfully runs the same DaCapo benchmarks that HotSpot JVM is able to run, resulting in **100% pass-rate**. Figure 5.15 shows the performance difference of HyperscaleJVM in comparison to HotSpot JVM on the Oslo testbed (Numascale platform) and Figure 5.16 plots the same numbers normalized on HotSpot JVM. The slowdown we observe using Hyperscale JVM ranges from 0.74 (speedup) to 4.22, while **the geometric mean of the slowdown we observe is 1.90**. Both JVMs fail to execute benchmarks *batik* and *tomcat*.

⁴⁰ <https://github.com/bee-hive-lab/Maxine-VM>

⁴¹ <https://github.com/JikesRVM/JikesRVM>

⁴² <http://dacapobench.org/>

⁴³ <http://ldbcouncil.org/benchmarks/snb>

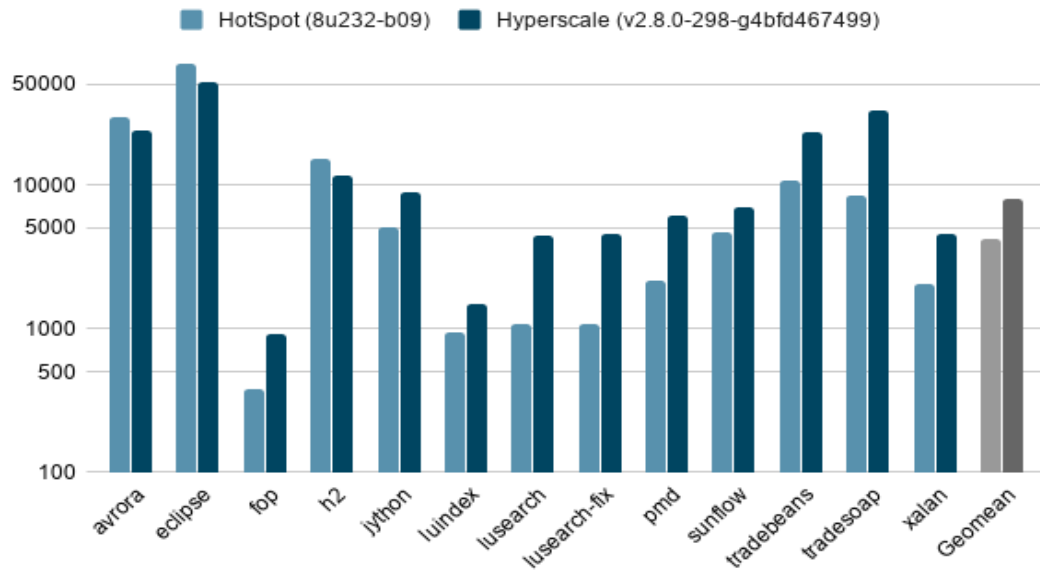


Figure 5.15: Hyperscale JVM vs HotSpot VM using DaCapo-9.12-MR1-bach on Oslo testbed.

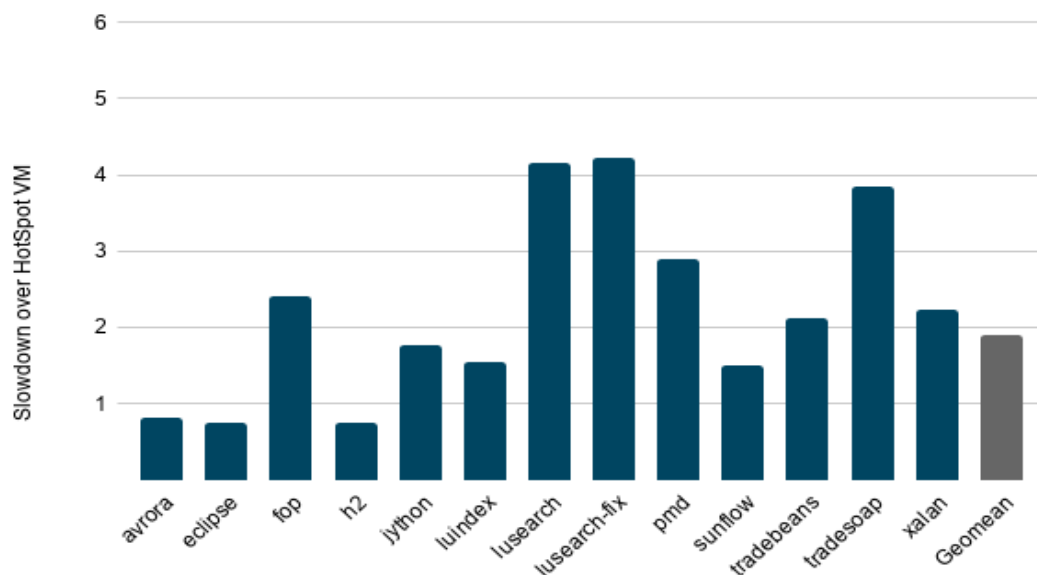


Figure 5.16: Hyperscale JVM slowdown over Hotspot VM using DaCapo-9.12-MR1-bach on Oslo testbed.

LDBC SNB on Neo4j v3.5.4

When running LDBC SNB with Neo4j v3.5.4 on HyperscaleJVM we observe **100% pass-rate**. Figure 5.17 plots the throughput of different queries for both virtual machines and Figure 5.18 plots the same numbers normalized on HyperscaleJVM. The throughput we observe with the HyperscaleJVM is up to 6 times less (Q6), while in the case of (Q8) 1.5 higher than that of Hotspot. **The geometric mean of Hyperscale JVM's slowdown over Hotspot** (regarding throughput in ops/sec) **is 2.48**. We attribute the large variation between the different queries to the different “choke points” of each query⁴⁴. Each query is essentially designed to stress different parts of the

⁴⁴ LDBC SNB Specification: http://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf#appendix.A

database engine (Neo4J in our case) and thus different parts of the JVM. Inspecting the *choke points* of the profiled queries we find that Q8 and Q13 (the two queries that perform better on the HyperscaleJVM) are the only queries in our tests to exercise choke points CP-5.3: “[QEXE] Intra-query result reuse” and CP-8.6: “[LANG] Handling paths”, respectively.

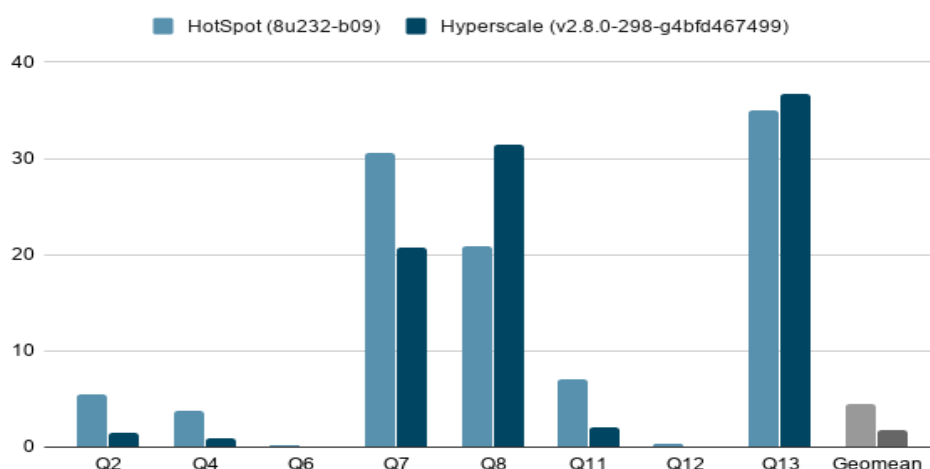


Figure 5.17: Hyperscale JVM vs HotSpot VM using LDBC SNB with Neo4j on Oslo testbed.

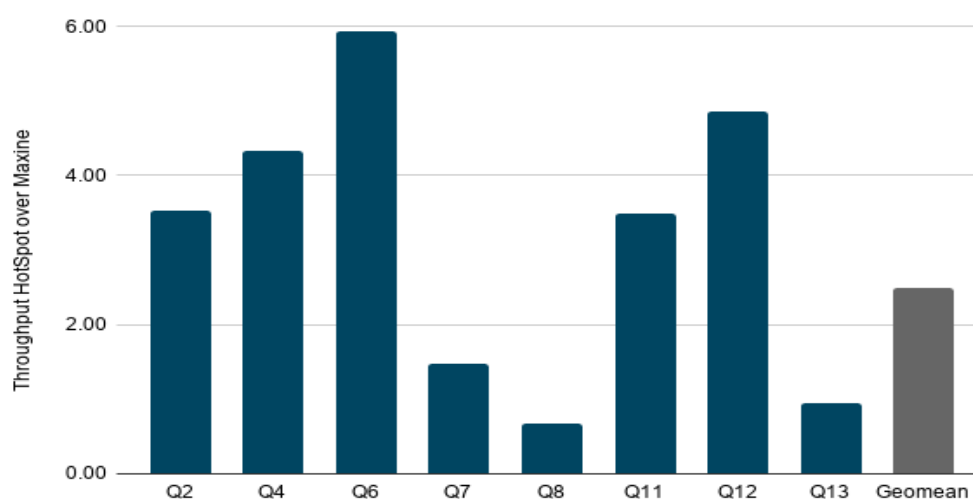


Figure 5.18: Hyperscale JVM slowdown over Hotspot VM using LDBC SNBwith Neo4j on Oslo testbed.

Manchester testbed

DaCapo-9.12-MR1-bach

Hyperscale JVM on the Manchester testbed successfully runs the same DaCapo benchmarks that HotSpot JVM is able to run except for *eclipse*, resulting in **90% pass-rate**. Figure 5.19 shows the performance difference of HyperscaleJVM in comparison to HotSpot JVM on the Manchester testbed (KMAX platform) and Figure 5.20 plots the same numbers normalized on HotSpot JVM. The slowdown we observe using Hyperscale JVM ranges from 1.12 (12%) to 4.16, while **the geometric mean of Hyperscale JVM’s slowdown over HotSpot is 2.7**. Both JVMs fail to execute benchmarks *batik*, *tomcat*, *tradebeans*, and *tradesoap* on the Manchester testbed.

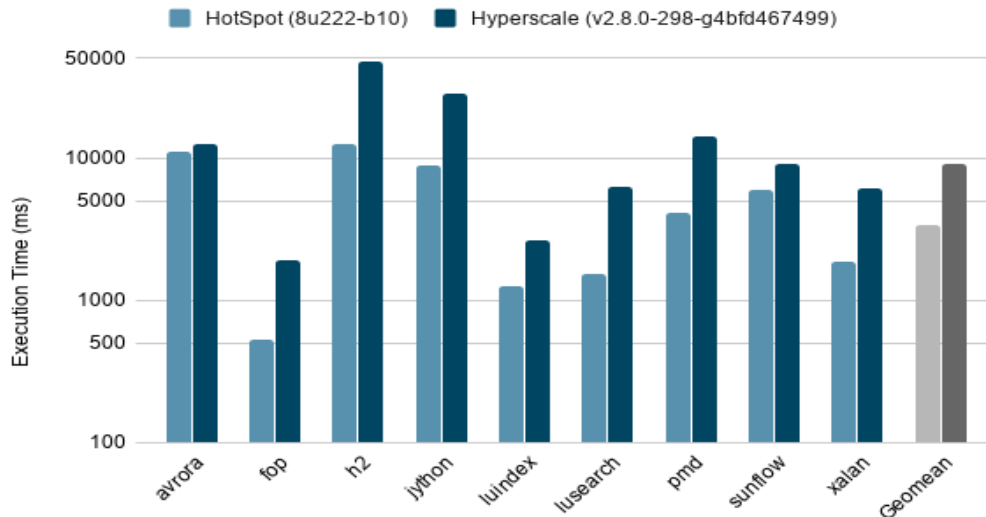


Figure 5.19: Hyperscale JVM vs HotSpot VM using DaCapo-9.12-MR1-bach on Manchester testbed.

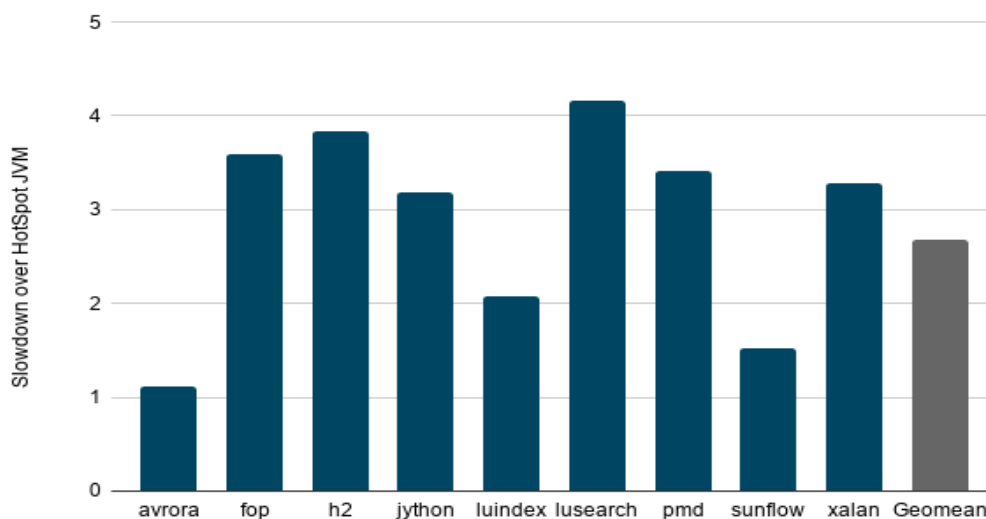


Figure 5.20: Hyperscale JVM slowdown over Hotspot VM using DaCapo-9.12-MR1-bach on Manchester testbed.

5.4.4 Performance evaluation against JikesRVM, the competing research VM

Hyperscale JVM is **the only currently available research JVM** that among others offers:

- Java 8 support
- AArch64 support
- High pass-rates on standard Java benchmarks

all of which were realized as part of the ACTiCLOUD project.

Despite not offering the above, the Jikes Research Virtual Machine (JikesRVM in short) is the only research JVM that currently competes with the Hyperscale JVM. As an indication of how popular

both VMs are, they both gather about 210 stars on their respective github repositories^{45,46}, with Maxine gaining all of its stars during the ACTiCLOUD project (since previously it was hosted on a different platform). The main reason that JikesRVM is still preferred by some over Hyperscale JVM for research is the Memory Management Toolkit (MMTk) that comes with it. MMTk provides the necessary building blocks for developing garbage collection (GC) algorithms without having to write boilerplate code. MMTk however is designed to be modular and not JikesRVM specific. Taking advantage of its modularity in ACTiCLOUD we have done an initial integration of MMTk in the Hyperscale JVM aiming to make Hyperscale JVM even more complete as a research JVM.

Since JikesRVM does not support Java 8 we can only use it to run the DaCapo benchmark suite, and specifically version 9.12-bach (the latest version working with Java 7). Neo4j v3.x requires Java 8, while v4.x requires Java 11. As a result JikesRVM cannot be used to run Neo4j. Additionally, since JikesRVM does not support the AArch64 architecture it is also impossible to use it on the KMAX platform. This leaves us with a single evaluation configuration, which is the DaCapo benchmark suite running on the Numascale testbed. Figure 5.21 plots the execution time on the y-axis for each benchmark in the DaCapo-9.12-bach suite, except for batik, eclipse and tomcat, that fail with the latest Java 8 Runtime Environment (JRE). Figure 5.22 plots the speedup gained by using Hyperscale JVM over JikesRVM for the same configuration. We observe that Hyperscale JVM outperforms JikesRVM in all benchmarks except for avrora where the performance of the two VMs is similar. **The geometric mean of Hyperscale JVM's speedup over JikesRVM is 1.94.** Apart from the performance difference it is worth noting that Hyperscale JVM is able to run the same benchmarks as HotSpot while JikesRVM fails to run fop, tradebeans, and tradesoap. That is, **Hyperscale JVM achieves 100% pass-rate** (using Hotspot as the baseline) while JikesRVM achieves 72%.

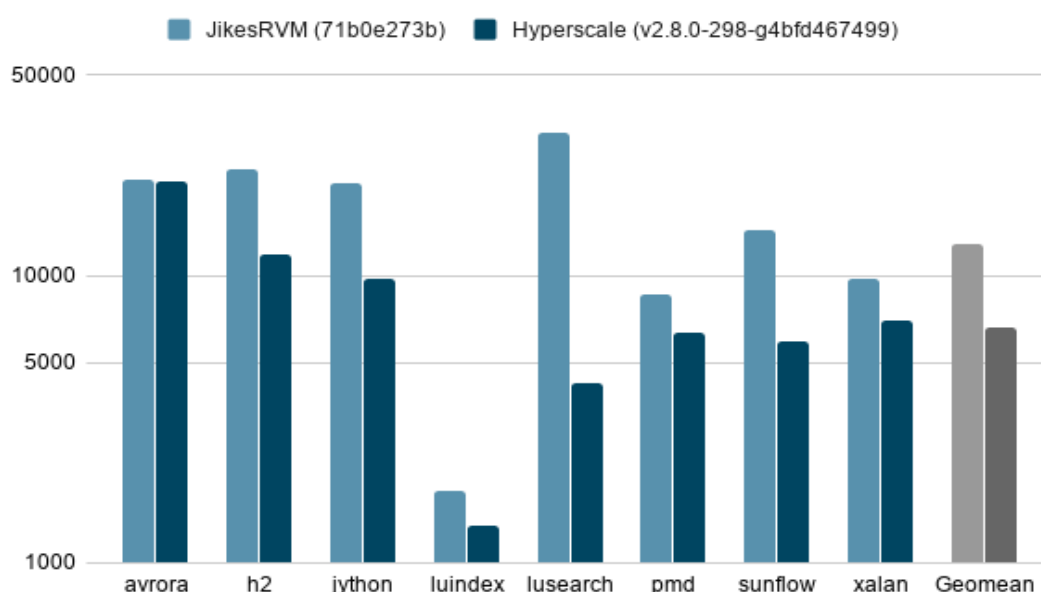


Figure 5.21: Hyperscale JVM vs JikesRVM using DaCapo-9.12-bach on Oslo testbed.

⁴⁵ <https://github.com/JikesRVM/JikesRVM>

⁴⁶ <https://github.com/bee-hive-lab/Maxine-VM>

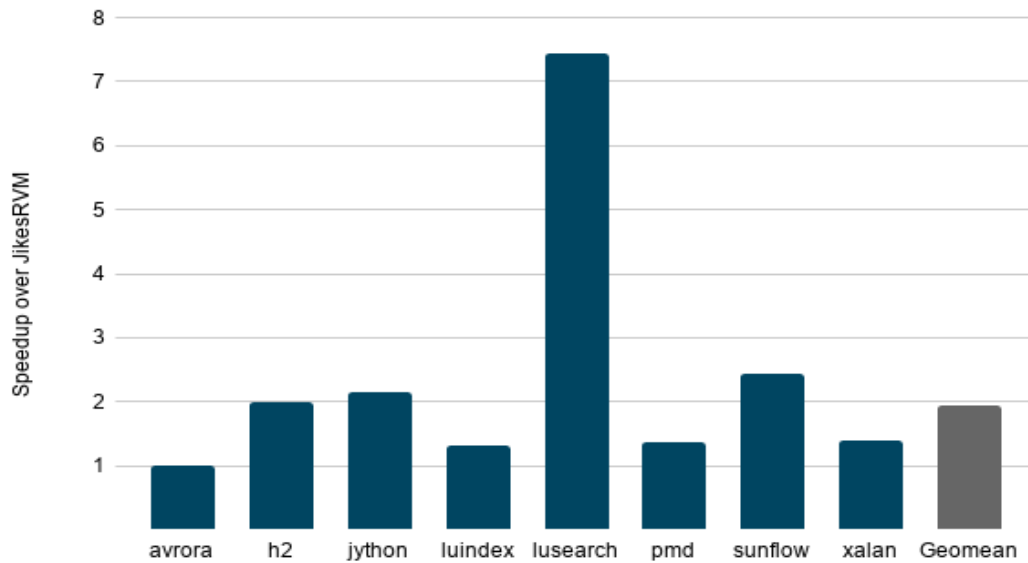


Figure 5.22: Hyperscale JVM speedup over JikesRVM using DaCapo-9.12-bach on Oslo testbed (fop, tradebeans and tradesoap fail on JikesRVM).

5.4.5 Memory Profiler Evaluation

In the context of the ACTiCLOUD project we have enhanced Hyperscale JVM with the ability to profile memory accesses and allocations of Java applications. Our work's main difference over prior work, like AntTracks VM⁴⁷, is that in our profiling we take into account the NUMA topology. HyperscaleVM's topology awareness enables the characterization of each memory operation as local or remote. This information can be utilized to attribute potential overhead to a specific cause (i.e., remote accesses). Hyperscale JVM can be configured to trace allocations, reads and writes. It can be configured to trace the whole application or specific parts of it as described in D3.6: *Hyperscale JVM v2.0*. In this section we first assert the memory profiler's correctness by comparing its results against those of AntTracks VM and using micro-benchmarks. Then we evaluate its performance, by measuring the overhead it adds to Hyperscale JVM when enabled.

Correctness

Hyperscale JVM supports two kinds of memory profiling, (i) allocation profiling and (ii) memory access profiling. To assert the correctness of allocation profiling we use the AntTracks VM as the baseline, while for the memory accesses profiling we developed a micro-benchmark that allows us to control the number of writes and reads it performs in order to cross-check them against the profiler results.

Allocation Profiling

To assert the correctness of Hyperscale JVM's allocation profiling we use the DaCapo benchmark suite and compare the results we obtain using Hyperscale JVM with those we obtain using AntTracks VM. Hotspot's Escape Analysis optimization reduces the number of heap allocations and therefore makes the AntTracksVM (which is based on Hotspot) results not directly comparable with Hyperscale JVM's (which doesn't perform escape analysis). To make the AntTracks results comparable to those obtained using Hyperscale JVM we disable the Escape

⁴⁷ http://mevss.jku.at/?page_id=1592

Analysis optimization of the first (using `-XX:-DoEscapeAnalysis` option). Figure 5.23 plots the results of our experiment. The geometric mean of the absolute difference between the Hyperscale JVM and the AntTracks VM results is 4.32%. We attribute this variation to the fact that Hyperscale JVM being a metacircular VM (i.e., it is written in Java itself) performs some heap allocations for its own needs, while HotSpot relies on libc instead of the Java heap for such JVM-internal allocations, thus Hyperscale JVM in most cases allocates slightly more objects than HotSpot.

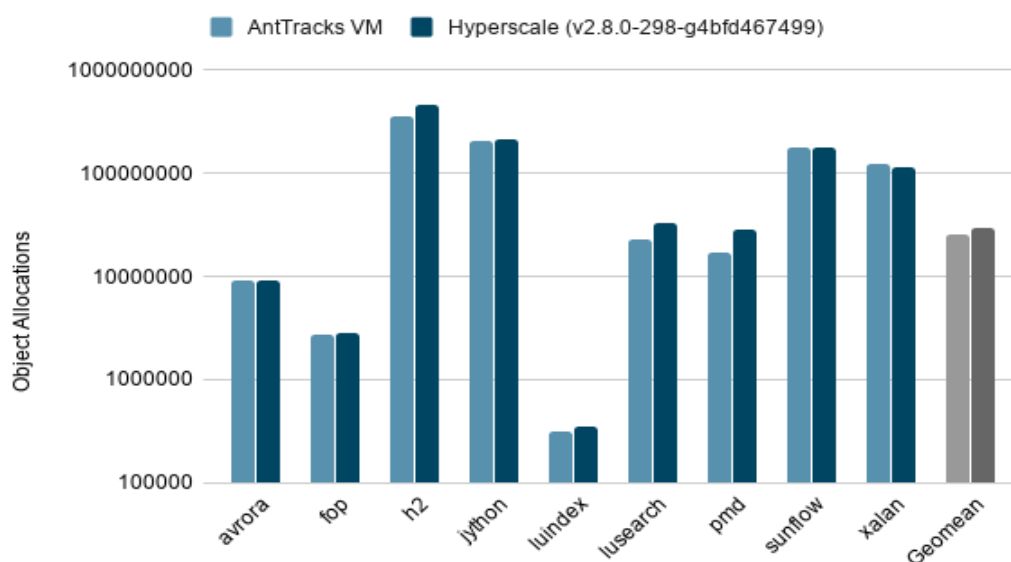


Figure 5.23: Number of object allocations per benchmark, as measured by AntTracks VM and Hyperscale JVM.

Memory Access Profiling

Regarding the correctness of the memory access profiling of Hyperscale JVM, since AntTracks VM does not offer this kind of profiling we opted to develop a custom micro-benchmark that generates a predefined number of memory accesses and use that instead. The micro-benchmark supports different kinds of accesses to test each of the Java byte-codes that result in a memory access, including array and field accesses. To avoid measuring noise, we perform a number of warmup runs before profiling the micro-benchmark. This way we avoid profiling JVM-internal operations which create memory accesses related to initializations, compilations, etc. and that are unrelated to the application. Note that since Hyperscale JVM is metacircular (i.e., written in Java itself) when profiling it also captures its own allocations. For memory accesses we can avoid this since the JVM code is compiled ahead of time and we have control over whether to add profiling instrumentation or not. On the contrary, since allocations are dynamic, there is no way to exclude JVM-triggered allocations from our measurements. As a result, Hyperscale JVM successfully captures the number of predefined allocations plus a small number of extra allocations. Those extra allocations are the result of some JVM-internal procedures that are inevitable no matter how long we warm up the JVM. Overall the obtained results indicate that the memory access profiling is accurate.

Overhead

Both allocation profiling and memory accesses profiling can be enabled or disabled when using the Hyperscale JVM. As profiling is based on code instrumentation a special build of Hyperscale JVM is required to support profiling. In this build, on each Java heap allocation, read, or write

Hyperscale JVM checks whether the corresponding event needs to be profiled or not (according to the profiling policy). As a result even when not actively profiling, those checks incur some overhead. To evaluate the overhead of the Hyperscale JVM memory profiler we run the DaCapo-9.12-MR1-bach benchmark suite with three different configurations. First we run with no profiling support (this is our baseline), then we run with profiling support but without profiling anything, and last we run with profiling support and profile the whole run (this is the worst case). Figure 5.24 plots the results of our experiments with execution time on the y-axis. Additionally, Figures 5.25 and 5.26 plot the overheads (on the y-axis) of the two configurations over the baseline. The geometric mean of the observed overhead is 31% for the checks, and goes to 149% when profiling is active. It is worth noting that the overhead heavily depends on the nature of the benchmark. This is expected since the higher the number of memory accesses and allocations the higher the impact of the profiling overhead, as more events need to be traced. It is worth noting that for some benchmarks, i.e. *avrora*, *lusearch*, *lusearch-fix*, *pmd*, *tradebeans* and *tradesoap* the overhead of actively profiling is comparable to that of having profiler support but without profiling anything. On the contrary in cases like *fop*, *h2*, and *jython* the overhead of profiling results in an order of magnitude longer execution times.

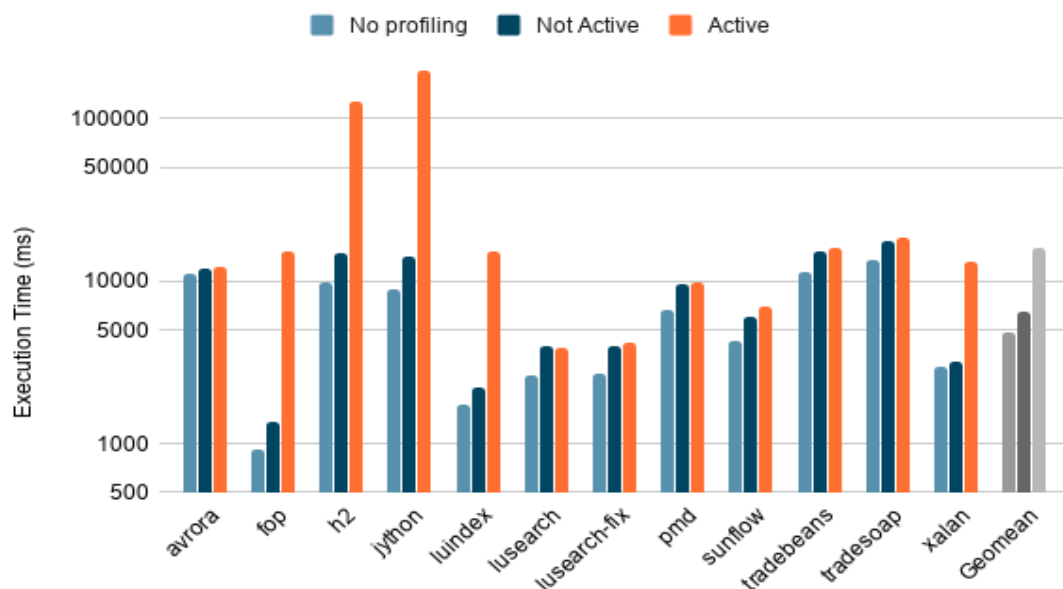


Figure 5.24: Comparison between Hyperscale JVM without profiling support (No profiling), with profiling support but no active profiling (Not Active), and with profiling support and active profiling (Active).

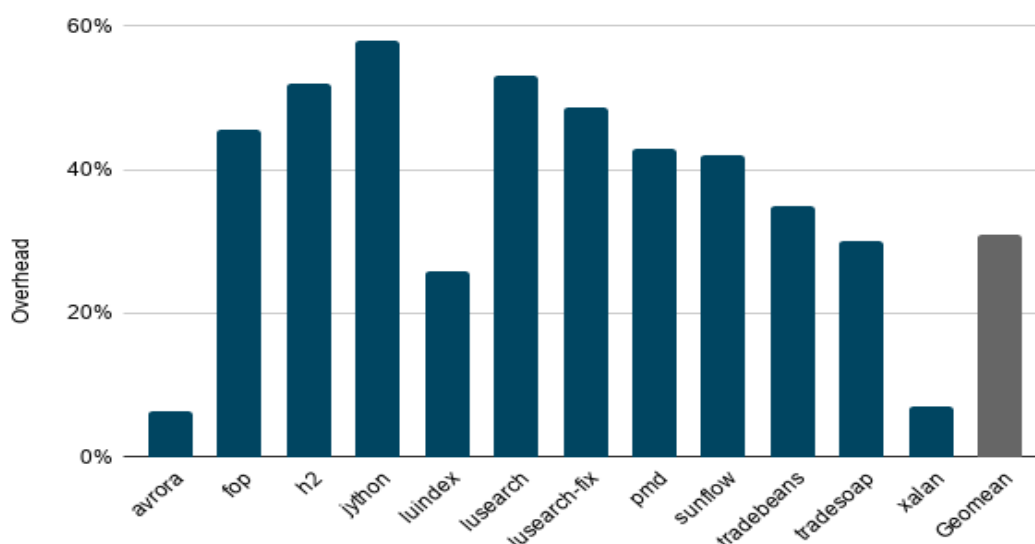


Figure 5.25: Overhead of Hyperscale JVM profiling support when not actively profiling.

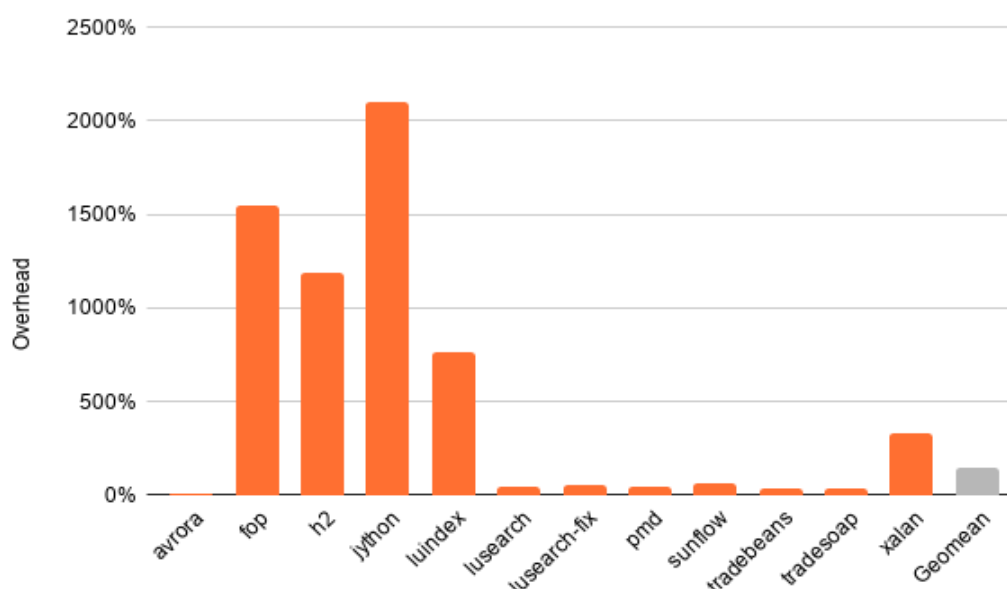


Figure 5.26: Overhead of Hyperscale JVM profiling support when actively profiling.

The high overhead of the profiler is something expected in general, given that the profiler interferes with very primitive events, i.e., reads, writes and allocations. To minimize the overhead during the application run Hyperscale JVM tries to only record events at runtime, and do the actual bookkeeping at safepoints, e.g. at garbage collection cycles. This still results in increased total execution times but the majority of the overhead comes from applications pauses, thus not affecting the application behaviour significantly (it might affect its responsiveness during garbage collection cycles though). Furthermore, since usually we are interested in profiling only a specific part of the code, e.g. the critical path, Hyperscale JVM, through its various profiling policies (see *D3.6: Hyperscale JVM v2.0*) enables users to limit the period of time that the profiler will be active, and thus also limit the overall overhead imposed on the application being profiled.

5.4.6 Conclusions

Overall, Hyperscale JVM is more feature-full than JikesRVM, offers Java 8 and AArch64 support, higher pass-rates and better performance. When compared with the standard OpenJDK JVM, HotSpot, on a single server Hyperscale JVM can be up to 5 times slower, but on average it is about 2 times slower. When compared on a large scale machine like the Numascale testbed in Oslo using more than one boards/servers HyperscaleJVM's performance is about half of that achieved by using the HotSpot VM. Although, this overhead is prohibiting for production code, it is an acceptable overhead to pay for a research JVM that enables NUMA-aware profiling and easy experimentation. Thus we believe that Hyperscale JVM can be used to increase productivity while creating prototypes, as well as to better understand Java applications' behavior on large ccNUMA systems using the NUMA-aware profiler. Ultimately, Hyperscale JVM can be the driving vehicle to achieve better scalability for Java applications on large scale systems.

6 Elasticity in Resource Provisioning

The development of the various components of the ACTiCLOUD architecture that provide scalability allow us to support resource elasticity as well. In this section we evaluate how the ACTiCLOUD architecture enables and supports elasticity for increased resource efficiency. We particularly focus on ACTiManager and the KMAX platform.

Methodology

We use the Kaleao's KMAX platform that is located in Manchester. We use 4 compute nodes of the KMAX system, with each node hosting 4 Exynos servers and each Exynos server is equipped with 8 cores, 4 big and 4 little cores.

Our setup is based on the OpenStack Pike version with libvirt and the KVM virtualization technology. We setup the OpenStack Controller on one Exynos server, and the OpenStack Telemetry services on another one in order to split the memory load between the two servers. Finally, we setup the other two Exynos servers as the compute nodes of our OpenStack installation. All of the Exynos servers run a modified version of Ubuntu 16.04 (Xenial), provided by KALEAO, which also includes a modified Linux kernel (v4.4.16). We installed: (i) the OpenStack Identity (Keystone) and Image Registry (Glance) services on the first Exynos, (ii) the Metering (Ceilometer) and Metric (Gnocchi) services on the second Exynos, and (iii) the Compute (Nova) and Network (Neutron) services, along the necessary Metering agents, on the other two Exynos servers of the KMAX node. Section 4 of the deliverable D4.4 "ACTiCLOUD Final Prototype" describes the necessary steps for integrating and setting up OpenStack and ACTiManager with KVM on top of the KMAX System.

Execution Scenario

In this scenario, we use a microbenchmark based on stress-ng⁴⁸ to demonstrate the synergy between ACTiManager and OpenStack elastic resource provisioning mechanisms on the ARM big.LITTLE KMAX platform.

We spawn a single-core (1 vCPU) virtual machine, which runs two stress-ng CPU stressors, in order to simulate a CPU-intensive workload, which needs to vertically scale in order to handle a spike in required performance. When the VM load average crosses a specific threshold, the VM issues a resize request to Openstack, to change its VM flavor from 1-vCPU to a 2-vCPU instance. The resizing mechanism is offered by the vanilla version of OpenStack.

ACTiManager detects the resize request, updates its internal state regarding the VM size, and runs both its external and internal placement logic to ensure that the requested resources are optimally allocated, taking into consideration the state of the resources and the other VMs that are currently running on the entire system.

Results

The following charts show the stress-ng performance, in (bogo)ops per second in four scenarios. A gold VM running stress-ng with 2 CPU stressors, with 1vCPU and 1GB RAM, which based on its load average requests from Openstack to resize itself, by switching to a flavor with 2vCPUs and 1GB RAM. The downtime in ops/sec between the flavor changes is due to the fact that Openstack has to restart the VM in order to complete the flavor change. Less intrusive mechanisms, like vCPU or memory hotplugging are also technically possible, but not supported by Openstack at the moment.

⁴⁸ <https://wiki.ubuntu.com/Kernel/Reference/stress-ng>

For the first scenario, we deploy ACTiManager, which pins the vCPU of the gold VM to a big core, achieving more than 50 ops/sec, detects the flavor change, and then pins both vCPU threads to separate big cores, ensuring optimal performance for the gold VM, achieving more than 100 ops/sec.

The second scenario shows the behavior of vanilla Linux on the KMAX platform. The vCPU initially is scheduled on a LITTLE core (it could have been scheduled on a big core but the decision is completely random) achieving ~25 ops/sec, and after the resizing operation is complete, the 2 vCPUs are scheduled on a LITTLE and a big core (again, the decision is random) achieving ~75 ops/sec, which results in suboptimal performance.

For reference, we also provide two charts of the microbenchmark behavior when the vCPUs are pinned to only big, achieving ~50 and ~100 ops/sec before and after VM resizing respectively, or LITTLE cores, achieving ~25 and ~50 ops/sec before and after VM resizing respectively.

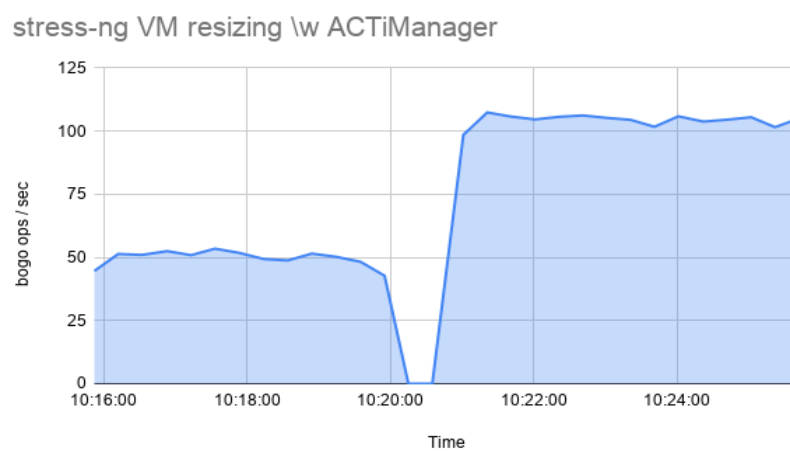


Figure 6.1: Resizing operation with ACTiManager.

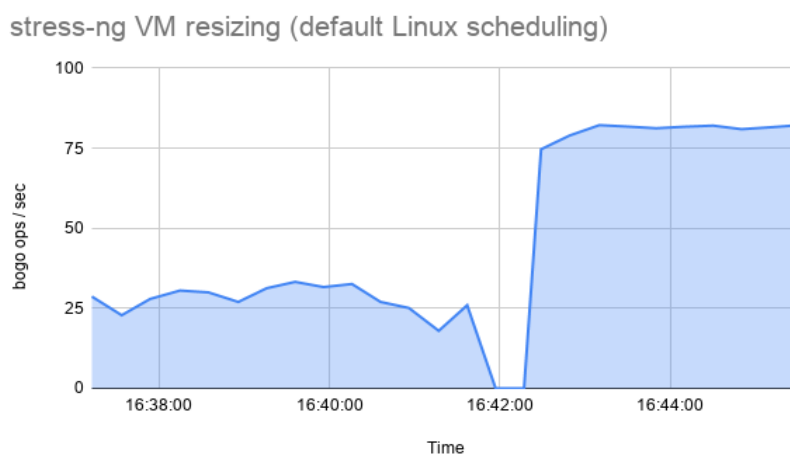


Figure 6.2: Resizing operation with default Linux scheduling.

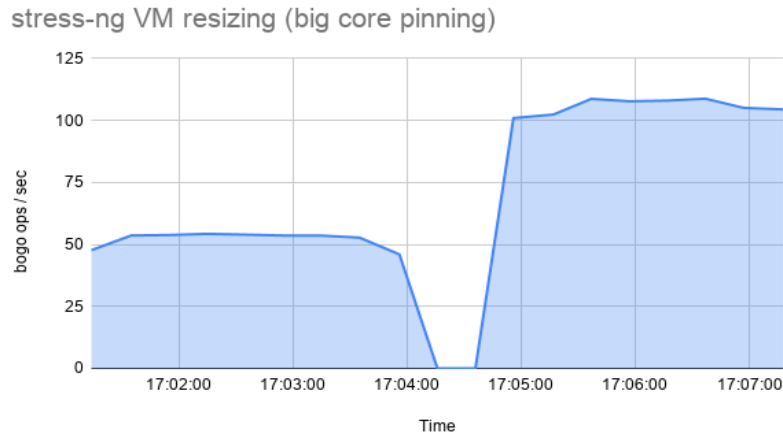


Figure 6.3: Resizing operation with static pinning on big cores.

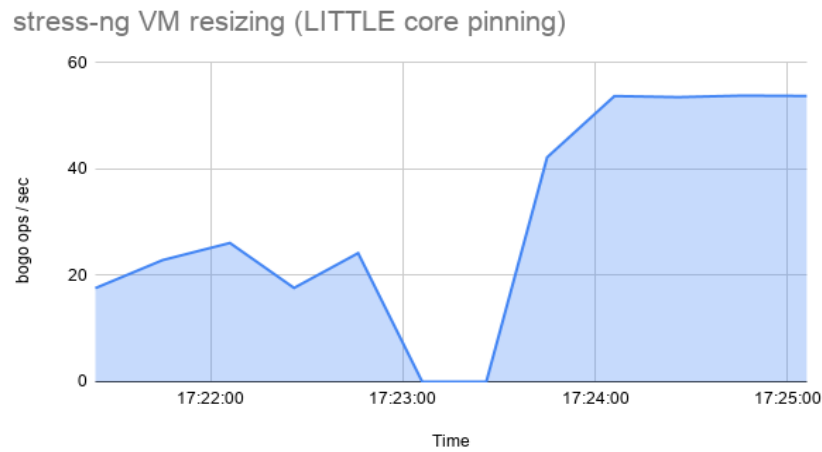


Figure 6.4: Resizing operation with static pinning on little cores.

Summary

Overall we observe that the ACTiCLOUD architecture efficiently supports elasticity for improved resource provisioning. We are in the process of evaluating a similar scenario based on a real world workload, specifically MonetDB and a more complex scenario involving multiple VMs and compute nodes as well as downscaling, to better showcase the benefits of the ACTiManager placement logic.

7 Conclusions

This deliverable describes the evaluation of the final ACTiCLOUD prototype and its constituent components. The evaluation is directly driven regarding the strategic objectives of the project and the use cases and business scenarios that the ACTiCLOUD architecture targets. Our evaluation results show that the final ACTiCLOUD prototype that involves contributions from project's partners across the entire computing stack manages to increase the effective utilization of resources by increased resource efficiency and performance stability, and to enable the deployment of resource demanding applications such as in-memory and graph databases in the cloud through support for scalability and elasticity in resource provisioning.