Future Generation Computer Systems 131 (2022) 194–208

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Effect of Hyper-Threading in Latency-Critical Multithreaded Cloud Applications and Utilization Analysis of the Major System Resources

Lucía Pons [a,*], Josué Feliu [a], José Puche [a], Chaoyi Huang [b], Salvador Petit [a], Julio Pons [a], María E. Gómez [a], Julio Sahuquillo [a]

[a] Department of Computer Engineering, Universitat Politècnica de València, Valencia, Spain
[b] Huawei Technologies CO., LDT., China

ARTICLE INFO

Article history:
Received 22 April 2021
Received in revised form 1 November 2021
Accepted 28 January 2022
Available online 2 February 2022

Keywords:
Cloud computing
Latency-critical workloads
Level of load
Tail latency
Resource sharing
Hyper-Threading

ABSTRACT

Multithreaded latency-critical applications represent an important subset of workloads running on public cloud systems. Most of these systems deploy powerful computing servers including Intel Hyper-Threading processors. Understanding how performance is affected by the consumption of the main system resources is a major concern for cloud providers in order to devise virtualization strategies that improve the system efficiency. With this aim, this paper first characterizes the impact of QPS on tail latency, analyzing different scenarios varying the number of threads and the thread-to-core allocation (single-task and multi-task execution) policy. The characterization study reveals that the performance of some applications does not scale with the number of threads, and the performance of some others is insensitive to the Hyper-Threading technology, so they can be allocated in less physical cores and improve system utilization. Identifying these applications, however, at run-time is challenging. Despite identifying these applications at run-time is challenging, this paper shows that they can be successfully detected at run-time by analyzing the utilization trend of the major system resources. In addition to CPU, we have also studied how assigning the share of each application of other major shared system resources impacts on performance. We outline considerations cloud providers should take into account to improve performance and resource utilization.

1. Introduction

In the public cloud the client has the illusion of accessing a rented machine with a given configuration. However, virtualization is used by cloud providers, which allows public cloud platforms to share the system resources among clients or tenant applications, typically hosted by virtual machines (VMs). This can be done provided that the service level agreement (SLA) specifying the target quality of service (QoS) is not violated. To deal with this concern, cloud providers need to understand how virtualization of the major resources of the cloud system affects tenant applications.

The complex infrastructure of cloud systems and the nature of cloud applications make it difficult to meet this concern. On the one hand, cloud systems are made up of multiple components that interact with each other, including server nodes, client nodes, storage nodes and the interconnection networks. On the other hand, among cloud applications, latency-critical applications deserve a special interest. Examples of these applications include many popular interactive services such as MongoDB [1] and NGINX [2], as well as image or speech recognition. Performance of these applications is usually given by the users' queries per second (QPS) the server is able to attend while guaranteeing QoS. Since QoS violations cannot be overlooked but could be hidden under the average latency metric, the key performance metric for these applications is the *tail latency*. This metric accounts for the latency of a small percentage of slowest requests, typically 95th or 99th percentile.

In order to perform representative cloud performance studies, the experimental platform should resemble as much as possible the real production environment. This fact is specially hard to achieve due to the complexity (both hardware and software) of production systems. Consequently, research works simplify the experimental framework. Unfortunately, existing studies omit important system components or do not consider the appropriate software environment, which results in losses of representativeness. For instance, in CoPart and Hypart [3,4] only LLC and main memory bandwidth are taken into account omitting network and disk resources. Similarly, Heracles and Parties [5,6] approaches do not study important resources like the disk. Regarding system

* Corresponding author.
E-mail addresses: lupones@disca.upv.es (L. Pons), jofepre@gap.upv.es (J. Feliu), jopucla@gap.upv.es (J. Puche), joehuang@huawei.com (C. Huang), spetit@disca.upv.es (S. Petit), jpons@disca.upv.es (J. Pons), megomez@disca.upv.es (M.E. Gómez), jsahuqui@disca.upv.es (J. Sahuquillo).

https://doi.org/10.1016/j.future.2022.01.025

software, recent works like [3,4] do not consider virtualization at all and other works like [6] use LXC containers, a more light-weight solution. In this work we implement the full system stack (hardware and software) to make the study representative.

This paper characterizes the multithreaded Tailbench [7] and *media-streaming* [8] applications in a production-like environment. Most CPU servers implement simultaneous multithreading (SMT) cores (e.g., Intel Hyper-Threading), which are seen as different logical CPUs by the OS. We study the impact of varying the number of threads or scalability, as well as the effect of Hyper-Threading (i.e., threads running on the same core), on the performance – tail latency and QPS while satisfying QoS – for each application. The results of this study reveal two main counter-intuitive findings regarding the insensitivity of applications to the number of server threads (non-scalable) and Hyper-Threading. Identifying these applications can help the cloud provider improve system performance and resource utilization. However, it is challenging to detect these applications at run-time. To this end, we study the correlation between these applications and the utilization of each major (CPU, LLC, main memory bandwidth, disk bandwidth, and network bandwidth) system resource showing that measuring the utilization and trend of these resources can help in the detection.

Finally, we study the impact of inter-VM interference at the main shared resources other than the CPU by applying existing technologies. We used Intel Resource Director Technologies (RDT) [9] to limit the LLC ways and allocate a share of the system memory bandwidth to the target VM; and a stressor microbenchmark that competes for disk bandwidth. Results show that scalable applications are more sensitive to LLC space and memory bandwidth while disk bandwidth consumption is strongly connected with disk applications performance.

All the studies are made from two main perspectives. On the one hand, research conclusions are highlighted in small *to take away* sections. On the other hand, practical actions oriented to the cloud provider are highlighted in small *cloud provider actions* sections that include hints to improve performance and the utilization at the major resources.

This paper makes three main contributions:

- We present two main findings: applications insensitive to the number of threads (non-scalable) and Hyper-Threading (SMT) insensitive applications. Notice that the latter finding advises cloud providers to assign threads of SMT insensitive applications on the same SMT core without adverse effects on performance.
- We study the correlation between the consumption of the major system resources and non-scalable and SMT insensitive applications. We show that these applications can be easily detected at run-time by the cloud provider as there exists a strong connection between resource consumption and the previous findings.
- We analyze the effect of inter-VM interference at the shared resources by limiting the available cache space, memory bandwidth and disk bandwidth to cloud applications. The study reveals that the performance of the studied applications can widely suffer when the share of a given resource reduces.

The remainder of this paper is organized as follows. Section 2 discusses the related work. Section 3 presents the hardware/software platform. Section 4 discusses key characteristics of latency-critical applications. Section 5 presents the workload characterization study. Section 6 relates the workload resource consumption with the main findings from the previous section. Section 7 presents a workload classification and analyzes the inter-VM interference at the major system resources. Finally, Section 8 presents some concluding remarks.

2. Related work

Due to the nature and fast evolution of the public cloud and related industry concerns, many research works have been proposed in the last few years. These studies apply a wide range of characterization methodologies that present important differences regarding virtualization levels (e.g., VMs, containers or no virtualization), considered performance metrics (e.g., execution time, throughput, or tail latency), target shared resources (e.g., LLC, main memory, etc.), and system configurations (e.g., SMT vs no SMT, system specifications). In contrast to previous research works, in this paper we perform a comprehensive study including all the main shared resources (i.e., CPU, LLC, main memory, network, and disk); focusing in a realistic configuration for cloud providers with a full virtual machine-oriented system stack and SMT-enabled CPUs; and considering tail latency as key metric to evaluate QoS from the tenant perspective.

The workload characterization performed in previous works is often leveraged to propose novel resource management approaches dealing with distinct shared resources that affect the behavior of these applications. A major difference among these approaches is whether the target workloads are representative of public cloud deployments. This is the case when the workload is implemented using virtual machines (VMs) running in full-stack system configurations. Consequently, this section summarizes previous research taking into account this differentiation.

2.1. Approaches not considering VMs

In [3], Park et al. focus on the interference at the LLC and/or memory bandwidth, but they do not consider other major shared resources in the cloud environment like the network or the disk. This approach characterizes the behavior of each application only according to the number of LLC misses and number of accesses per second. However, the system performance is not considered at all. Moreover, the approach mostly focuses on HPC workloads and only a small section studies the behavior of scenarios where a single latency-critical application runs concurrently with batch workloads.

Also regarding memory bandwidth, in [4], the impact of Intel Memory Bandwidth Allocation (MBA) technology on performance is studied. MBA provides a rather coarse interface to limit main memory bandwidth consumption. Therefore, this approach also studies complementary techniques such as thread packing [10] and clock modulation [11].

In [12], each application is categorized considering four main aspects that affect performance: scale-up (amount of resources per server), scale-out (number of servers per workload), server configuration, and interference (symbiotic workloads). This categorization can be used to establish the right amount of resources allocated to an application to reach a given performance level while maximizing overall resource utilization (i.e., avoid overprovisioning). The resources that are taken into account are number of compute cores, memory, and storage capacity. Unlike our work, neither LLC occupancy, main memory bandwidth, nor disk bandwidth are considered.

In [5], similarly to our work, Lo et al. characterize the impact of interference at shared resources on performance for different load levels; however, just three latency-critical Google workloads are characterized. To study the effect of interference, synthetic benchmarks stress each shared resource. Nevertheless, the interference at the disk is not analyzed.

In contrast, Parties [6] considers disk interference. In this work, six latency-critical applications are studied. However, Parties, as some of the works mentioned above, studies applications running in LXC containers [13] (i.e., Linux containers), which are more light-weighted that full VMs and thus, the studied workloads are not representative in a significant amount of public cloud deployments.

2.2. Full system stack approaches with VM support

In [14], a mix of batch (Hadoop running over Mahout and Spark jobs) and latency-critical applications (memcached jobs) are studied in three representative workload scenarios (minimum, medium, and high load variability). The focus of this study is to find the optimal mapping of applications to reserved and on-demand VM instances. This paper presents a high-level evaluation perspective. There is no insight about the exact resources (e.g., cache, disk bandwidth) that affect the jobs' execution time. This work only deals with VM instances whose size is defined in terms of numbers of virtual CPUs.

With the goal of determining the best VM to physical core mapping, in [15], VMs are classified as compute or memory intensive, considering readings from several performance counters. In particular, this work uses just three performance events (L2 cache misses, L1 cache misses, and committed instructions) to classify applications. The focus is on best-effort scenarios and the evaluated workloads are four-application mixes composed of SPEC CPU2006 benchmarks, which are scheduled in pairs to the processor cores.

Finally, DeepDive [16] pursues to identify interference between VMs in IaaS clouds. Deepdive uses about a dozen low-level metrics (including performance events acting at the L2 cache as the LLC in the system, the iostat and netstat tools, and available hypervisor -VM- statistics) to find out if interference is taking place as well as what is the main shared resource where interference rises. Nevertheless, DeepDive lacks from important performance events, which are present in modern processors as well as partitioning mechanisms (i.e., Intel CAT and MBA) dealing with shared resources like the LLC and the main memory bandwidth. The analysis introduces some unexpected metrics like the use of the private L1 cache misses, mainly because the L2 is the LLC. In modern processors, the latency of most L1 cache misses is hidden by the out-of-order mechanisms. Experiments use three cloud and latency-critical (web search) workloads.

3. Experimental system

This section presents a general overview of the experimental platform, both hardware and software, employed in this work and discusses the devised VM infrastructure.

3.1. System specifications

The experimental framework is made up of three physical servers (main, client and storage nodes) interconnected with two 20 Gbps dedicated links. Fig. 1 shows a block diagram of the experimental platform, including the three system nodes: main node, client node and storage node, and the installed packages and libraries. Below, the system nodes are described.

The main node is where the developed framework is installed, including the *Resource and Application Manager software* (ReAM). This node runs the VMs hosting the tenant applications or benchmarks to be studied. This two-socket node has two Intel Xeon Silver 4116 processors with six memory channels holding three 16 GB DDR4 DIMM each, which amounts to 96 GB (6 × 16GB) DRAM capacity and supports 115 GBps bandwidth. The Intel Xeon Silver 4116 processor is equipped with 12 cores and a 16.5 GB 11-way LLC. It supports Intel Cache Allocation Technology (CAT) and Memory Bandwidth Allocation (MBA) [17], allowing the partition of LLC space and memory bandwidth.

The client node is used to run the clients for the client–server benchmarks, e.g., applications from the TailBench and CloudSuite benchmark suites. A script in the client node is executed to launch the client side of each application.

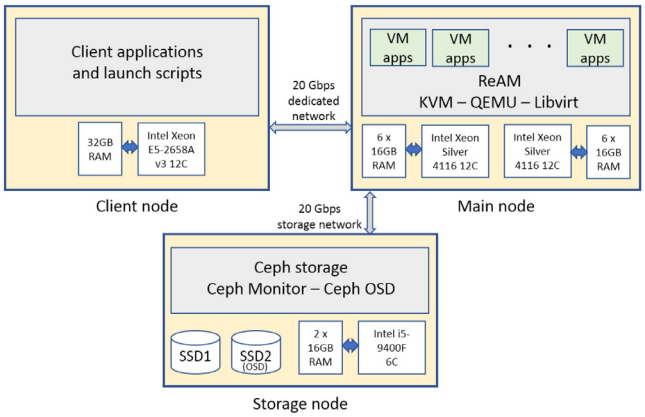


Fig. 1. Overview of the complete experimental framework.

The storage node implements a remote storage system with Ceph [18] (version 14.2.9). This node has a SATA SSD hard drive devoted to the remote storage. It runs a Ceph OSD daemon and a Ceph monitor. We would like to remark that, in this work, Ceph is not used to increase performance in a multi-node environment by leveraging Ceph's parallel I/O. Instead, we use Ceph to provide our system with a realistic software stack similar to those used by current cloud providers.

Regarding the system software, the installed operating system in the three nodes is Ubuntu 18.04 LTS. The main node has kernel version 5.4. On this node we developed the *Resource and Application Manager software* (ReAM), which is able to monitor resource consumption and assign fractions of the major system resources (e.g., number of CPUs, main memory space, LLC space, network and disk bandwidth) to each individual VM.

3.2. VM infrastructure

To create a realistic infrastructure resembling that of servers deployed by cloud service providers, we developed ReAM to be compatible with VMs. KVM [19] is installed as hypervisor, QEMU [20] (version 2.11.1) as virtualizer and libvirt [21] as virtualization manager (version 4.0.0). Additionally, the main node has a virtual switch configured to connect the physical network interfaces with the VMs. The virtual switch is configured using Open vSwitch [22] (OvS, version 2.9.5) and Data Plane Development Kit [23] (DPDK, version 17.11.9). OvS and DPDK enable direct transfer of packets between the user space and physical interface, bypassing the kernel network stack. This setup boosts network performance compared to the default packet forwarding mechanism implemented in the Linux kernel. This configuration is similar to that used by OpenStack on its compute nodes [24].

To reduce the start-up overhead, the designed framework makes use of the Ceph block device snapshots [25]. For each VM, we take a snapshot of the state where the OS boot process is already performed and it is ready to receive the command to launch the target benchmark.

3.3. Intel hyper-threading

The processor of the server node implements Intel Hyper-Threading technology [26], that is, Intel's implementation of the simultaneous multithreading (SMT) paradigm [27]. The key characteristic of SMT processors is the capability to issue multiple instructions from different threads on each cycle, which allows increasing the processor throughput. However, this means that instructions from the applications – or threads – concurrently running on the same core compete, among other shared

resources, for the scarce issue ports; thus, introducing inter-thread interference that harms their performance with respect to their individual execution.

Hyper-Threading processors typically support the concurrent execution of two threads. From the operating system perspective, a given physical core is seen as two logical cores or CPUs. In a multi-core processor with Hyper-Threading cores, it is important to differentiate between physical and logical cores when talking about resource sharing. Two threads running on different physical cores only share the off-core processor resources, mainly the LLC and main memory. However, when two threads are assigned to the two logical cores of the same physical core, they also compete for intra-core components. These components are critical for performance and include, among others, issue ports, reorder buffer (ROB), physical registers, load queue, store queue, functional units, as well as L1 and L2 caches. As a consequence, the performance of a given thread highly depends on the demands of the internal components of the thread it is co-running with. Previous works [28] have studied this performance drops. Appendix analyzes the impact of Hyper-Threading in different processor architectures.

4. Workload generation

4.1. Latency-critical applications

Latency-critical applications are widely used in cloud environments. In these applications, the cloud server typically implements online services (e.g., speech recognition or language translation) and must respond to the input requests within specific latency bounds to guarantee QoS and provide a satisfactory user experience.

As a representative set of latency-critical applications, we use the TailBench benchmark suite [7]. This suite includes eight representative workloads of today's latency-critical applications. For the sake of making this paper self-contained, we briefly describe the main characteristics of the studied applications:

- ***img-dnn*** is a handwriting recognition application based on OpenCV. The application uses randomly chosen samples from the MNIST database.
- ***masstree*** is a fast in-memory key–value store written in C++. Each user's request often involves many tens or hundreds of requests to the key–value store; therefore, it has very short latency requirements.
- ***moses*** is a statistical machine translation system written in C++. It is driven using randomly-chosen dialogue snippets from the opensubtitles.org English–Spanish corpus.
- ***shore*** is a transactional on-disk database. It uses the industry-standard OLTP benchmark TPC-C. Its database and logs are both stored in a solid state drive.
- ***silo*** is an in-memory transactional database designed for modern multicores. It uses TPC-C like *shore*, although they differ significantly in how they store and access data.
- ***specjbb*** is an industry-standard Java middleware benchmark. Java middleware is widely used in business services and must often satisfy strict latency constraints.
- ***sphinx*** is a compute-intensive speech recognition system written in C++. Speech recognition systems are an important component of speech-based interfaces and applications such as Apple Siri, Google Now, and IBM Speech to Text.
- ***xapian*** is a search engine written in C++ that is widely used both in software frameworks (e.g., Catalyst) and popular websites (e.g., the Debian wiki).

Cloud providers often need to evaluate the efficiency shown by servers that stream multimedia contents. However, none of the studied Tailbench applications exhibit such behavior since all of them present negligible network demands. Because of this reason, other applications outside the Tailbench benchmark suite need to be considered. With this aim, this work also analyzes the ***media-streaming*** workload from CloudSuite [8]. This application, based on the NGINX web server, is a streaming server that hosts synthetic videos of various lengths and qualities. The server is accessed by clients based on the *httperf's wsesslog* session generator, which performs a set of requests per session for videos stored in the server. This popular application in today's data centers makes it possible to further broaden the spectrum of studied behaviors.

4.2. Setting representative load levels

Unlike HPC benchmark suites, TailBench applications include the QPS parameter that allows generating a wide range of load levels. For experimental purposes, this parameter needs to be tuned to match the desired workload level. Below, we discuss the approach followed to obtain a representative range of QPS values for each of then studied application.

To simulate real client–server behavior, clients of TailBench applications issue requests to the server following the Zipfian[1] distribution [29,30]. To do so, the clients use a *request generator* to indicate the points of time when requests are issued to satisfy the demanded QPS and the Zipfian distribution. Unfortunately, when a large QPS is demanded, it may happen that the client cannot generate and send the requests fast enough. In this scenario, the client might still reach the desired QPS on average but break the Zipfian distribution, which compromises the representativeness of the experiment.

To address this issue and make experimental results representative, we defined the metric *timely requests ratio*, which accounts for the percentage of requests that the clients are able to issue fulfilling the request times generated following the Zipfian distribution. A request is defined as *non-timely* when the request time, provided by the request generator, is earlier than the current time. Notice that clients can break the distribution and still meet the requested QPS on average. In this regard, we found that even though a single client can generate up to thousands of requests per second, usually it can only generate between 400 and 600 without breaking the distribution. To provide representative results, we checked that the target QPS is achieved while guaranteeing that, at least, 97.5% of the requests are timely.

Under this condition, we explored the QPS range of each workload as well as the required number of clients to generate the requests. QPS steps were chosen to cover a wide range of representative CPU loads, from 10% to saturation (over 50%). This allows to study the behavior at a usual average utilization [31] (e.g., 20%) and how it is affected as the load grows. Table 1 presents the results. As observed, QPS widely varies among applications, ranging from 0.2 to 8.5 in *sphinx* and from 250 to 4000 in *silo*.

Unlike TailBench applications, the load of *media-streaming* is indicated as the number of *sessions*. Any number of clients can be launched to complete the target number of sessions. In this work, we explore the client load up to a maximum of 70 sessions and 24 clients, since we found out that a higher number of sessions yields saturated results.

[1] The Zipfian distribution is a related discrete power law probability distribution that states that, for a given set of items (e.g., words of a text), the frequency of any item is inversely proportional to its rank in the frequency table.

Table 1
QPS range and number of clients used for each workload to guarantee that the ratio of timely requests is above 97.5%.

Workload	QPS range	Number of clients
img-dnn	100–3100	12
masstree	250–2500	12
moses	10–1100	6
shore	10–2000	12
silo	250–4000	18
specjbb	250–3500	18
sphinx	0.2–8.5	2
xapian	100–2800	4

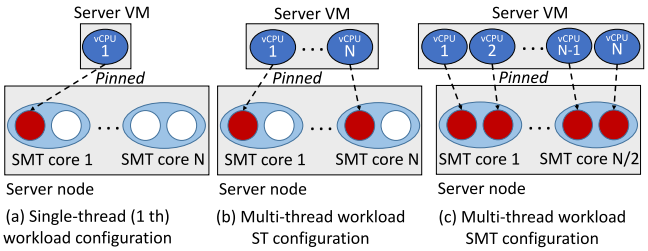


Fig. 2. Server VM core configurations studied. *N* stands for the number of threads spawned by the server, each assigned to a different vCPU.

5. Workload characterization

Due to the multithreaded nature of many cloud workloads and the Hyper-Threading technology of many cloud servers, countless scenarios regarding the server configuration can be studied. This section first presents the studied scenarios and performance metrics. After that, we analyze the results of each metric across the different scenarios.

5.1. Studied server scenarios

We characterize the behavior of Tailbench applications and *media-streaming* across different server configurations or scenarios. Scenarios have been designed to evaluate both the effect of Hyper-Threading and the scalability with increasing number of threads the server application spawns. We assume that each thread is assigned to an individual virtual CPU (vCPU).

Three main configurations have been considered as shown in Fig. 2 to generate scenarios of interest. In configuration (a), only one vCPU is used, pinned to a logical core, which is in charge of running the single-threaded server application. In configuration (b), the server application runs *N* threads (assigned to *N* vCPU cores), and each one runs in single-task mode in a different physical core.

Finally, configuration (c) differs from configuration (b) in that the *N* threads (vCPU cores) are pinned in pairs to the two logical cores of the same SMT physical core, running both threads concurrently. Notice that scenario (c) uses half the number of SMT cores of scenario (b).

Under these configurations, N has been evaluated both for two and eight threads or vCPUs, providing five main scenarios. More precisely, configuration (b) splits into scenario *2-ST* (2 threads are assigned to 2 cores working each in single task mode) and *8-ST*, and configuration (c) breaks down into *2-SMT* (2 threads are pinned to half the number of cores working in multi-task mode) and *8-SMT*. Notice that configuration (a) can be seen as a particular case of configuration (b) for *N* equals to 1, so it will be referred to as *1-ST* and represents the fifth scenario. These scenarios allow to compare the behavior of a single-threaded server against a multithreaded server, as well as the impact of Hyper-Threading in a relatively high multithreaded (i.e., eight threads) server.

5.2. Studied performance metrics

In order to characterize the workloads in the introduced scenarios we need to define the metrics of interest. The characterization studies presented in this work cover both performance and resource utilization. Below, we list the studied metrics.

- **95th tail latency**, i.e., the 95th percentile of the *sojourn* (end-to-end) times, which considers both queue and service times. This percentile indicates that only 5% of the responses take longer to complete than that value. In case of *media-streaming*, it shows the 95th percentile of the server response time. In this paper, we consider that this metric defines the QoS (see Section 5.3).
- **CPU utilization**. This metric refers to the average CPU utilization of the logical cores assigned to the VM (vCPU cores), e.g., for the 2-thread scenarios, we report the average utilization of the two CPUs where the application is running. To obtain the utilization of each CPU, we use the data collected from the file /proc/stat which reports statistics about the kernel activity aggregated since the system first booted. The CPU utilization quantifies the fraction of time the application is running on the processor (i.e., percentage of active time with respect to the total time). Notice that this time also accounts the time the application is waiting for main memory accesses even though the processor is stalled and no instruction can be executed.
- **LLC occupancy** and **main memory bandwidth** show the sum of the LLC capacity occupied and main memory bandwidth consumed, respectively, by the vCPU cores.
- **Network bandwidth**. Two main bandwidths can be considered from the server (i.e., running-VM perspective): received and transmitted. The latter is dominant and the former is almost negligible, so the study focuses on the latter one.
- **Disk bandwidth**. This metric refers to the disk bandwidth consumed by the running VM at the server node.

Next we present and analyze the results for each metric.

5.3. QoS and tail latency analysis

5.3.1. Defining QoS

Nowadays, many online-service workloads present tail latency QoS constraints. These constraints are part of the Service Level Agreement (SLA) in some cases; in other cases, users should make sure of hiring enough resources to meet their target QoS. To determine realistic tail latency QoS constraints in our experimental setup and evaluate whether the workloads meet the QoS, we define the QoS requirement for each workload as a function of the average service time. This approach is based on the one proposed by Delimitrou et al. [32]. The QoS target for each workload is defined as 5× the average service time achieved with a CPU utilization of 20% in the 1-ST scenario. We will refer to this value as **LQoS**. Experimental results in this paper assume that QoS is met whenever the 95th tail latency is lower than the corresponding LQoS.

Table 2 presents the LQoS values (in ms) and the QPS supported by the single-threaded server for each workload that meets the QoS latency constraint. Tail latency requirements range from 0.5 ms for *silo* to 4275.4 ms for *sphinx*. Notice that the LQoS of *sphinx* is over 4 s, which may seem rather high in comparison with other speech recognition services. However, we found that

these values are in line with the results of this application presented by other researchers [7]. Finally, in *media-streaming* LQoS is defined in terms of 95th percentile of the response time [33]. Many web and streaming services are time-bounded, requiring the response time to be less than a fixed threshold [34]. In this work we have set the threshold to 500 ms as the maximum assumable delay the user should wait for the demanded streaming contents.

5.3.2. Tail latency analysis

This section studies the effect of the studied configurations on the tail latency provided that LQoS is met. Fig. 3[2] shows the results. The figure consists of nine plots, one for each studied application. In each plot, the LQoS value is represented by an horizontal dotted line. Since all the studied metrics experience high variation, scatter plots are used instead of line plots. More precisely, the LOWESS [35] algorithm has been used to create a smooth trend line to ease the analysis.[3] In the analysis of this metric, we found a major observation and two main findings discussed below.

Observation 1: Performance Scalability

It is expected that the performance of the studied multi-threaded latency-critical applications, which are build to execute in cloud systems, scales with the number of threads. This is true for most applications, as they exhibit a great performance scalability and support higher QPS, while meeting LQoS, with 2 and 8 threads than when running on the single-threaded server. Two main tendencies can be identified in *thread scalable* applications.

Firstly, applications that significantly improve the supported QPS with both 2 and 8 threads. Applications showing this behavior are *img-dnn*, *moses*, *sphinx* or *xapian*. For instance, *img-dnn* meets LQoS with 600 QPS, 1600 QPS and 2900 QPS in the 1-ST, 2-ST and 8-ST scenarios, respectively. This means that the supported QPS in the 2-ST and 8-ST is 2.67× and 4.83× higher than in the 1-ST.

Secondly, applications that show minor QPS improvements with 2 threads but exhibit a high performance increase with 8 threads. This is the case for *shore* and *media-streaming*. For the latter application, it can be observed that the response time in both 8-ST and 8-SMT scenarios does not saturate in the same way as the other applications. The response time grows up to about 75 sessions, point after which it remains constant and starts decreasing. Even though the LQoS point has not been reached, this trend indicates that the server has saturated.

Finding 1: QPS Insensitive Applications to the Number of Server Threads

Many factors such as the application configuration, the application version, the system's features, and the virtualization environment affect scalability. Therefore, adding a high number of server threads does not always translate into the server being capable of supporting more QPS. We found three applications showing this behavior. Below we analyze the reason behind this unexpected behavior for each application.

Specjbb supports up to 700 QPS with the single-threaded configuration and improves up to 1200 QPS with the 2-ST scenario, but then experiences a little drop to 1100 QPS with the 8-ST server that quadruples the number of threads. The reason behind this behavior is, as introduced before, that the version of *specjbb* provided in the Tailbench benchmark suite has a fix number of warehouses (a unit of stored data) during the whole run, and there is a one-to-one mapping between warehouses and threads, meaning that the number of server threads cannot be configured as in other applications. This makes the configuration of this application not optimal for scalability.

Table 2
Tail latency QoS (LQoS) requirements for the Tailbench workloads in our experimental platform.

Workload	Tail latency QoS	QPS single-thread
img-dnn	3.6 ms	600
masstree	1.4 ms	1000
moses	7.1 ms	250
shore	25 ms	90
silo	0.5 ms	700
specjbb	0.7 ms	1500
sphinx	4275.4 ms	0.6
xapian	6.2 ms	300
media-streaming	500 ms	25

Masstree also supports more cumulative QPS with 2 threads than with 8 threads. We looked into the reasons that explain this behavior and we found that the DRAM fetch cost is a limiting factor in *masstree's* scalability as also found in recent research [37]. More precisely, the number of processor stalls due to main memory accesses rises with the number of contending queries (QPS), thus the off-chip DRAM bottlenecks the performance and no improvement is obtained when increasing the number of threads. As a consequence, the CPU utilization drops in the highly-threaded scenarios as studied next.

Silo also shows the best results in the 2-ST scenario, but followed by the 1-ST scenario, which outperforms the 8-ST server. However, there is not a consensus on the behavior of this application. For instance, in [38] *Silo* is identified as scalable, and in [7] authors argue that the overhead of adding threads prevents this application to scale beyond a few threads.

Finding 2: SMT Insensitive Applications

Since our server implements Intel processors with Hyper-Threading technology (that is, the codemark used by Intel to refer to its SMT implementation), an important decision to improve the server efficiency is whether threads should be pinned to the same physical core (i.e., to two logical cores of the same SMT core) or they should be pinned to distinct cores. In case threads are pinned by couples to physical cores, only half the number of SMT cores need to be used. However, as explained in Section 3.3, running two threads concurrently in the same core causes interference at the shared core-resources, harming the performance (i.e., QoS) with respect to single-task execution. This concern is analyzed next.

As expected, it can be observed in Fig. 3 that for a given number of threads, the supported QPS is in general higher in the ST scenarios. For instance, in *sphinx* a significant difference rises between SMT and ST configurations, which translates into 2× and 1.4× higher QPS in the 2-ST and 8-ST scenarios over the corresponding SMT scenarios, respectively. The general trend is that applications achieving higher performance in ST scenarios are those identified above showing performance scalability with the number of threads. We use the term *SMT sensitive* to refer to these applications.

Nonetheless, it can be noticed that some applications do not benefit at all from having the threads pinned to separate cores and working in single-task mode. We refer to these applications as *SMT insensitive*. This is the case for the three insensitive applications to the number of threads, which show barely any difference between the 8-SMT and 8-ST servers since their performance does not scale. Counter-intuitively, the scalable applications *media-streaming* and *shore* also show a SMT insensitive

[2] Notice that due to QoS requirements range from milliseconds to seconds, different scales have been used to ease the analysis.

[3] We leveraged an existing implementation [36] of LOWESS, and set the alpha parameter to 0.6 and the polynomial degree to 1.

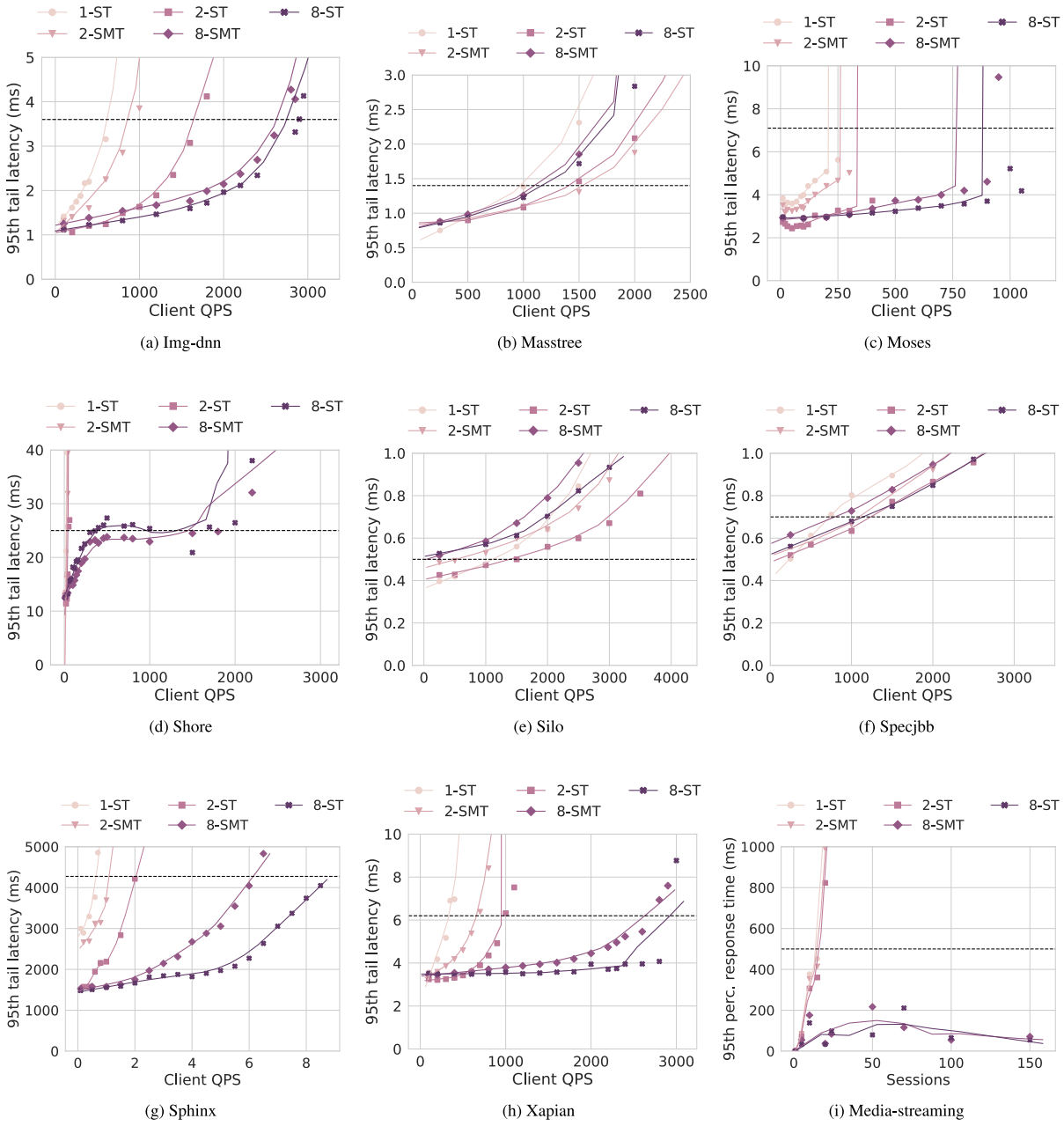


Fig. 3. 95th tail latency (ms) for Tailbench applications and 95th percentile of the response time (ms) for *media-streaming*.

behavior. We looked into this event and found that the main reason behind this behavior is that the CPU is not a critical resource in these applications. In other words, reducing the pressure in the CPU resources does not translate into actual performance gains.

An interesting observation is that in *masstree* (Fig. 3(b)) the 2-SMT configuration outperforms the 2-ST configuration (i.e., using only one core). We investigated this unexpected behavior and found that *masstree* keeps all the data (i.e., key–value database) in memory. These data are shared among threads. Thus, pinning two server threads to the same core reduces the core resources for each thread but improves the data sharing through the private L1 and L2 caches.

To Take Away. *In Finding 1, we have identified three workloads that do not experience scalability when running in 8-threaded scenarios, and five applications showing different degrees of scalability. In Finding 2, we have identified that there is a strong connection between scalability and SMT sensitive applications. In general, more scalable applications support higher QPS when running in single-task mode while low scalable applications present a close SMT insensitive behavior.*

Cloud provider actions for performance improvements. *Cloud provider admins could improve resource utilization and take benefit from Finding 1 by constraining to two logical cores those server applications presenting poor scalability. The key question is how to detect these applications. Regarding Finding 2, important CPU utilization savings can be achieved in case of SMT insensitive applications are detected at run-time.*

The major challenge for cloud providers lies on how applications can be identified as (i) QPS insensitive to the number of server threads (Finding 1) and/or as (ii) SMT insensitive (Finding 2) at production time. To deal with this challenge we looked into the behavior exhibited by applications by measuring the utilization of major system components. The main purpose of this study is to analyze possible correlations between the resource

consumption of the major system components and both QPS and SMT sensitivity of applications.

6. Major system resource consumption analysis and findings' correlation

This section analyzes the utilization of the major system components (CPU utilization, LLC occupancy, main memory trend, network and disk) for each workload, and establishes logical relationships between the consumption of these resources and the findings presented in the previous section.

6.1. Analysis of CPU utilization

The CPU utilization was measured at specific client load levels of interest that can be identified in Fig. 3 as follows. It was studied at abscissa of the point where the tail latency curve crosses the LQoS horizontal line, which represents the client load (i.e., QPS or number of sessions in *media-streaming*) that meets the LQoS for each studied scenario.

Fig. 4(a) shows the results. It plots the quartiles of the average CPU distribution with *box plots* for each scenario. Central quartiles are represented as boxes, variability outside the boxes as vertical lines, and outliers as points. A strong relationship can be observed between the CPU utilization and the findings discussed above.

Regarding Finding 1, the three applications showing no scalability with the number of threads experience a sharp drop in the CPU utilization in the highly threaded scenarios. On the contrary, this sharp drop is not experienced in applications showing scalability. Among these applications, those presenting high CPU utilization with 1-ST (e.g., *moses* and *sphinx*) further increase their utilization with 8-ST. For instance, *sphinx* increases its CPU utilization over 90%. The reason is that high CPU utilization is caused by *memory stalls* that block the processor execution for longer, as further discussed in the next section. In contrast, those presenting medium CPU utilization (e.g., *img-dnn* and *xapian*) do not experience a CPU utilization increase in the 8-threaded scenarios, but it remains equal or even decreases.

Regarding Finding 2, SMT insensitive applications present a low CPU utilization (e.g., below 20%) regardless of the number of threads, with the exception of *masstree* in the Tailbench applications, which presents a medium (by 40%) CPU utilization for the low-threaded scenarios. As discussed above, *masstree* presents this particular behavior because the off-chip DRAM main memory bottlenecks the system performance. Thus, adding more server threads translates into a slight decrease in the overall performance and a drop in the average CPU utilization since there are 4× more threads providing less system performance. In contrast, SMT sensitive applications present a medium to high CPU utilization in the low-threaded scenarios.

6.2. LLC occupancy and main memory bandwidth

The shared Last Level Cache (LLC) can have a high impact on the system performance [39,40]. Some recent processors implement hardware support that allows the system administrator partition this component among applications at run-time (e.g., at the granularity of OS quantum). This section analyzes the space requirements of each application and its trend when increasing the number of threads. Below, we analyze the results, mainly focusing on the correlation with Observation 1.

On the one hand, applications present both different cache requirements per individual query and number of QPS; consequently, the LLC occupancy experiences wide differences among the studied applications. On the other hand, for a given application, the higher the amount of queries the server processes, the higher the LLC occupation regardless of the number of threads. However, only scalable applications can significantly rise the LLC occupancy in the 8-threaded scenarios. It is also worth to mention that the main memory bandwidth consumed depends on the number of LLC misses. Therefore, it is strongly connected to the data locality of the cached blocks as well as the LLC occupancy, since a high occupancy would rise in many LLC capacity misses.

Figs. 4(b) and 4(c) present the distribution across all the execution quanta of the LLC space and main memory bandwidth consumed, respectively, by the applications in the studied scenarios at the LQoS threshold. Taking into account both LLC occupancy and main memory bandwidth, two main types of applications can be observed related to Observation 1 and Finding 1 discussed below:

(a) Applications where the 8-threaded scenarios significantly rise the supported QPS. In these applications, the huge increase in the supported QPS translates into a huge increase in the LLC utilization, which makes these applications consume nearly all the cache capacity (over 14MB out of 16.5MB). Exceptions are *shore* (though this application consumes 12MB), and *media-streaming*, whose average LLC occupancy is around 7 MB but in some execution phases of the 8-threaded scenarios, it manages to occupy the full cache space. This common behavior is exhibited by the Tailbench applications showing scalability with 8 threads. Notice that memory bandwidth increases in a 10× factor, which means that the LLC is suffering a high amount of cache misses. These misses translate into long main memory latencies that introduce significant processor stalls, rising the CPU utilization. Additionally, the relative difference in bandwidth is much higher (log scale) than the difference in LLC space, meaning that most cache misses are capacity misses.

(b) Applications showing small differences in the cache occupancy regardless of the number of threads. This behavior can be observed in the remaining applications (i.e., those not showing QPS scalability) like *masstree*. The reason is that QPS is not improved in the 8-threaded scenarios. Consequently, the main memory bandwidth among the studied scenarios also experiences minor differences.

Almost all the applications exhibiting the former behavior nearly consume all the cache space in the 8-threaded scenarios suffering a high amount of cache misses. This means that the performance of these applications is clearly limited by this processor component, hence the cloud provider should identify these applications in order to assign them more space. Section 7 analyzes this claim in further detail.

6.3. Disk bandwidth

Among the studied Tailbench applications only three of them are disk oriented: *moses*, *shore* and *xapian*. The remaining applications make a scarce usage of the disk. In fact, their disk bandwidth consumption remains below 1 MB/s during most of the execution time. Fig. 4(e) presents the results.

Disk bandwidth is not constant along the execution time but it experiences peaks and drops along time. As it can be observed in Fig. 4(e), most applications have many outlier values (represented by diamond-like dots), meaning that disk bandwidth is usually low and presents some peak values at few intervals of time. In contrast, more disk-consuming applications like *shore* include less outliers as the median of the disk consumption (horizontal line crossing the boxes) is much higher.

In spite some Tailbench applications are disk oriented, the presented results make the disk bandwidth consumption not a concern in our experimental platform in terms of scalability. That is, this resource does not prevent the performance of disk oriented applications from growing in the 8-threaded scenarios. However, remember that even if the SSD installed in the

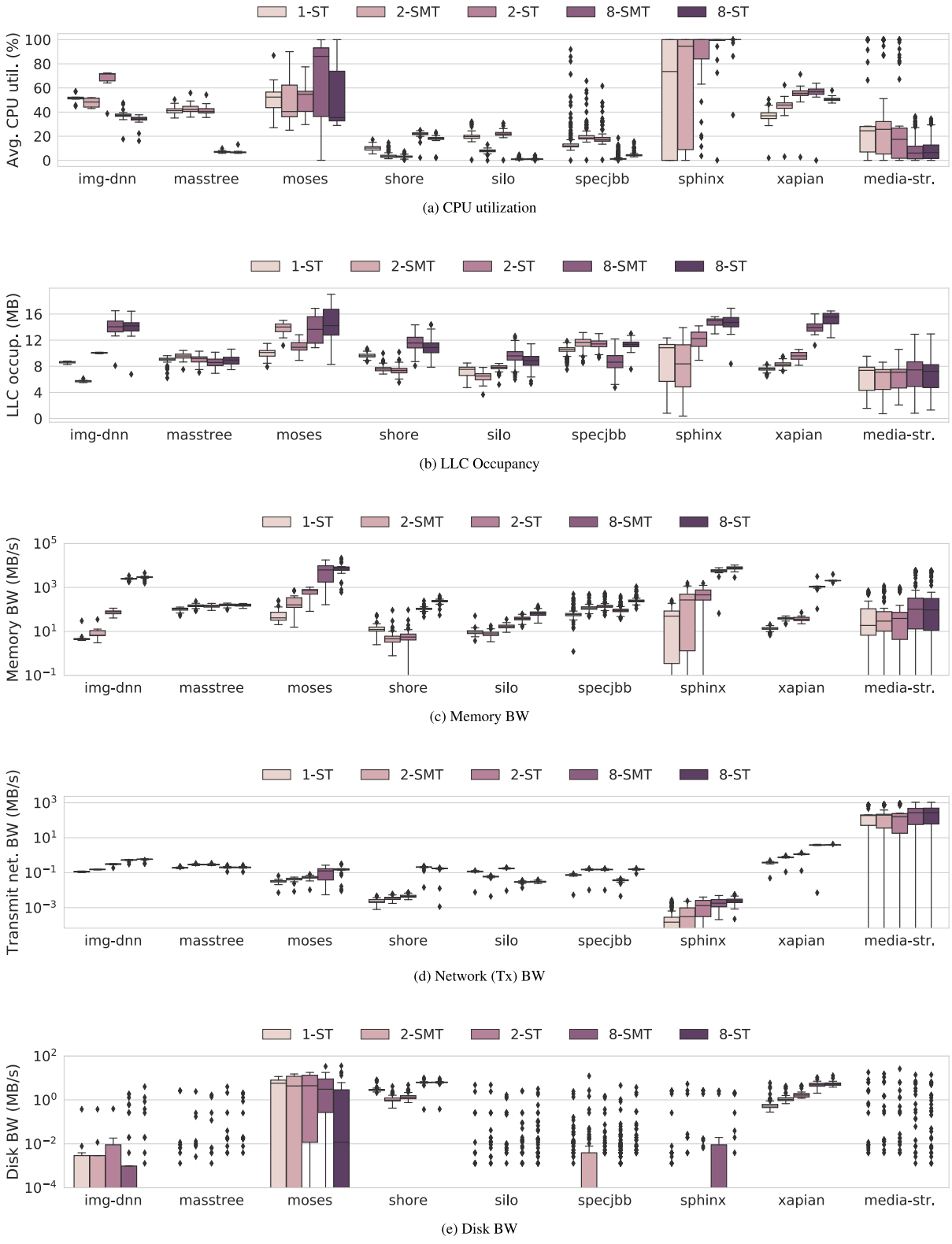


Fig. 4. Box plots showing the resource consumption supported by the studied applications in each scenario before reaching LQoS.

storage server allows up to 500 MB/s in large sequential reads, the bandwidth is significantly reduced when operations become small, random, and reads and writes are combined. Therefore, interference is likely to take place when multiple applications perform disk I/O operations at the same time. This issue will be studied in more detail in Section 7.4.

To Take Away *In this section we have analyzed and identified the relationship between the findings and the utilization of the main*

Table 3
QPS scalability, SMT sensitive, and resource consumption of the studied applications. Resource consumption (low, medium or high) is measured with the single-threaded server and the trend (↓ or ↑) represents the behavior change from 1-ST to 8-ST. Conditions to fulfill each finding are highlighted in yellow (Finding 1), blue (Finding 2) and green (Findings 1 and 2).

Workload	Resource Utilization					Finding 1 Affinity	Finding 2 Affinity
	CPU utilization	LLC Occupancy	MM trend	Network Bw	Disk Bw	QPS Scalability with # 8 threads	SMT sensitive
img-dnn	Medium ≈	Medium ↑	↑ ↑	Low	Low	+ +	+
masstree	Medium ↓	Medium ≈	≈	Low	Low	No	No
moses	High ↑	High ↑	↑↑↑	Low	Medium	+	+
shore	Low ↑	Low ↑	↑	Low	Medium	+ +	No
silo	Low ↓	Medium ≈	≈	Low	Low	No	No
specjbb	Low ↓	High ≈	≈	Low	Low	No	No
sphinx	High ↑	Medium ↑	↑ ↑ ↑	Low	Low	+ +	+ +
xapian	Medium ≈	Medium ↑	↑ ↑	Low	Medium	+ + +	+ +
media-streaming	Low ↓	Low ≈	↑	High	Low	+ +	No

system resources. We found that both non-scalable and SMT insensitive (e.g., silo) applications present medium to low CPU utilization in the low-threaded scenarios, that drastically drops in the 8-threaded scenarios. The term low and high are used to refer to below and above average, respectively. On the contrary, both scalable and SMT sensitive applications (e.g., moses) experience a medium to high CPU utilization in the low-threaded scenarios that tends to increase in the 8-threaded scenarios. These applications present a significant increase in the main memory bandwidth consumption as the number of threads increases.

Cloud provider actions to detect applications. *In order to help cloud providers, we present Table 3 that shows the average utilization (low, medium and high) of the studied resources for 1-ST, and the trend (upwards or downwards arrow) it experiences with 8-ST. The table also shows the main memory bandwidth behavior with 8-ST over 1-ST. With ↑↑↑, ↑↑, and ↑ denotes that memory bandwidth rises with 8 threads by 10^4, around 10^3, around 10^2, respectively. This table summarizes the previous findings. The conditions to fulfill Finding 1 and Finding 2 are highlighted in yellow and blue, respectively. The conditions that remain valid for both findings are colored in green. It can be concluded that only checking the CPU utilization and the memory trend is enough for cloud providers to identify the applications with higher resource requirements and carry out the corresponding actions to improve resource management previously described.*

6.4. Network bandwidth

The studied TailBench workloads present negligible (i.e., below 1 MB/s) network bandwidth requirements. Thus, this consumption is not a concern in our experimental 20 Gb/s network. In contrast, *media-streaming* consumes much higher network bandwidth. Fig. 4(d) presents the experimental results.

In *media-streaming*, however, the bandwidth consumption is a major concern. Similarly as discussed above regarding disk bandwidth, network bandwidth experiences high peaks and drops as sessions are dynamically started and finished. This can be observed in Fig. 4(d), where the lower whisker of all *media-streaming* plots drops down to 0 MB/s and the upper whisker reaches around 1000 MB/s.

To Take Away *The network allows cloud tenants to access the cloud system. This resource is typically overdimensioned since queuing delays in this component can force the cloud system violate the QoS regardless of the improvement actions made in the other major system components.*

7. Analysis of inter-VM interference at the main shared resources

So far, we have analyzed the performance of the server workloads by allocating threads in pairs to the same physical core or each thread to a different core. This section goes a step beyond and pursues to analyze the impact of the inter-VM (i.e., inter-workload) interference at the main shared system resources, other than the CPU. To this end, we focus on resources whose sharing can be controlled by the cloud provider. In other words, the cloud provider can apply existing advanced technologies to allocate certain amounts of specific shared resources to the running applications.

Some examples of these technologies are Intel Resource Director Technology (RDT) [17], which implements Cache Monitoring Technology (CMT) and Cache Allocation Technology (CAT) that allow monitoring and partitioning the LLC occupancy, respectively; and Memory Bandwidth Monitoring (MBM) and Memory Bandwidth Allocation (MBA) from Intel RDT that support monitoring and partitioning the memory bandwidth, respectively. Below, we use all these technologies to limit the available space or bandwidth for the target application, and study the effect on performance [3,4,39–41]. Notice that by limiting the amount of a given resource we mimic an scenario where the remaining fraction of the shared resource is being used by other VMs – from either the same or distinct tenant – competing for that resource.

Before analyzing the impact, we first categorize applications from the major system resource that constraints its performance.

7.1. Workload classification

The performance of the studied workloads is mainly dominated by a major system resource (CPU, disk or network). The studied workloads present a diversity of behaviors covering all the major system components. This section relates applications with the major consumed resources and the discussed findings.

CPU Workloads. This group includes workloads mainly dominated by CPU resources, including both core, LLC and DRAM memory, and make a negligible use of network and disk. DRAM memory is included as CPU since, from the OS perspective, the time the CPU is waiting for DRAM accesses in accounted as CPU utilization. One could expect that CPU workloads scale in performance. However, this is not always the case since a well balanced design and a 95th tail latency large enough are required . Instead, CPU applications present both scalable and non-scalable behaviors. Examples of scalable workloads are *img-dnn* and *sphinx*, and example of non-scalable *masstree*, *silo* and *specjbb*. The former set of applications behave as SMT sensitive and the latter as SMT insensitive.

Disk Workloads. Applications in this group present a significant disk bandwidth consumption that makes them stand out from the remaining categories. Two main categories can be distinguished according to whether the dominant bandwidth is incurred by disk read or write operations. Examples of applications presenting a medium disk utilization mainly due to read operations are *moses*

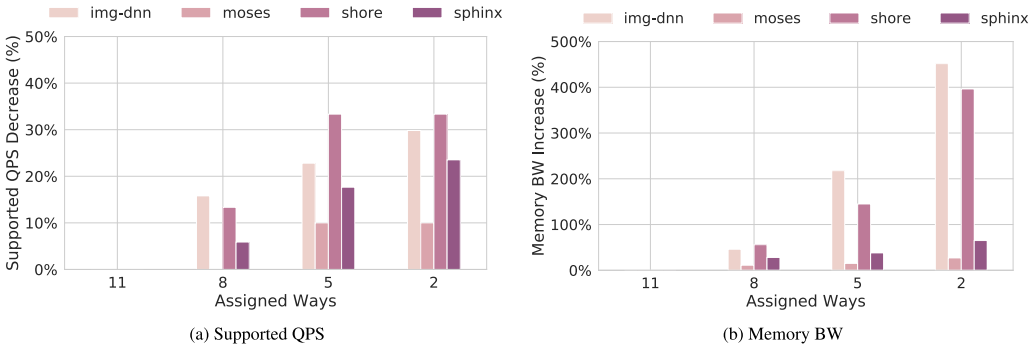


Fig. 5. Impact of limiting the number of LLC ways on the maximum supported QPS and memory BW.

Table 4
Workload Classification.

CPU		Disk		Network
Non-Scalable	Scalable	Read	Write	Streaming
specjbb silo mastree	img-dnn sphinx	moses xapian	shore	media-streaming

and *xapian*, while *shore* is an example of workload presenting a medium disk utilization due to writes. An interesting observation is that all these applications scale in performance with the number of threads; however, each of them presents a different behavior with respect to SMT.

Network Streaming Workloads. As mentioned above, *media-streaming* is the only studied workload presenting a noticeable network bandwidth consumption. This application scales with the number of threads, increasing the memory bandwidth. As the number of threads increases, the CPU utilization reduces as much as 10%.

Table 4 presents the devised groups (first row), categories (second row) and workload classification. Notice that all the applications except a subset of CPU workloads scale their performance as the number of threads increases. Next, we study the impact on performance of constraining the LLC, main memory bandwidth, and disk bandwidth.

7.2. LLC partitioning analysis

This section analyzes the impact of reducing the LLC space available to the target server application. Intel CAT supports assigning specific cache ways to applications. To this end, different Classes of Service (CLOS) are defined with specific cache ways. Then, each target workload (or VM) is associated to a CLOS.

The Intel Xeon Silver 4116 processor, used as server in the experimental platform, implements a 11-way 16.5 MB LLC, hence each cache way provides 1.5 MB storage capacity. To study the impact of reducing the available cache capacity to the target application, we reduced the number of cache ways assigned to the application to 7 ways (i.e., 12 MB), 5 ways (7.5 MB) and 2 ways (3 MB). The remaining cache space is assumed to be assigned to other VMs running on other cores and competing for this resource.

The study analyzes four workloads: two CPU scalable workloads (*img-dnn* and *sphinx*) and two disk workloads (*moses* and *shore*) on the 8-ST server. Fig. 5(a) presents the percentage decrease of the maximum supported QPS (w.r.t. the QPS achieved with full cache space) when varying the number of LLC ways assigned to each workload. Limiting the LLC can translate into an increase in the main memory BW, especially in memory intensive applications. This side effect can be appreciated in Fig. 5(b) which shows the memory BW increase. As observed, the applications showing highest scalability, *img-dnn* and *shore*, present bigger QPS reduction (over 30% with 2 cache ways), and consequently, the bus bandwidth consumption significantly rises in a factor over 3.5×. Despite that *moses* and *sphinx* also present important scalability and cache occupancy, they experience lower QPS reduction than the other applications. This is due to the fact that *moses* and *sphinx* consume much more memory bandwidth under no cache constraints.

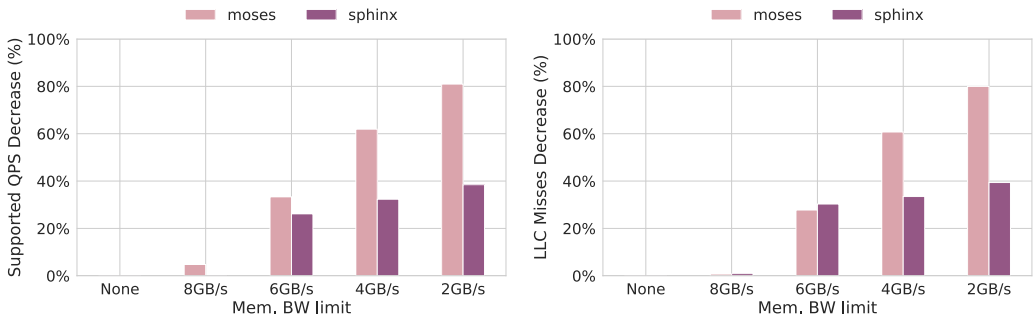
Cloud provider actions for performance improvements. *Results show that applications presenting a high scalability are much more sensitive to the available cache space. Limiting this space translates into an important rise in the memory bandwidth of these applications. Therefore, the cloud provider should consider both QPS scalability and memory bandwidth as a guide to limit the cache ways assigned to a given VM.*

7.3. Main memory bandwidth analysis

The impact of memory bandwidth on performance depends on the LLC demands of the applications and the underlying system organization. For instance, a huge LLC can catch most of accesses of some memory hungry applications, hence reducing the number of off-chip memory accesses. To carry out this experiment, we used Intel Memory Bandwidth Allocation (MBA), which works similarly to Intel CAT; applications are assigned to CLOSes since we can only limit the amount of memory bandwidth that the CLOSes can use.

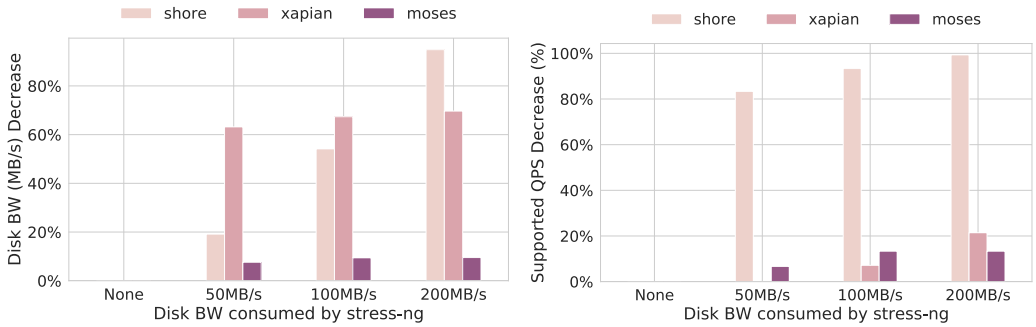
For illustrative purposes, we considered the same four applications as in the previous section and we studied the effect of reducing the memory bandwidth to 8-, 6-, 4- and 2-GB/s on both the supported QPS and effective bandwidth consumption. The huge LLC catches most of the working set of non-memory hungry applications when they run alone on the system and the whole cache is available for them. Therefore, we do not analyze those applications (*img-dnn* and *shore*) having a MPKI (misses per kilo instructions) of the LLC lower than 0.4 since they are scarcely or not affected at all by constraining the memory bandwidth under these conditions.

Fig. 6(a) shows the results for *moses* and *sphinx*. As observed, both applications suffer a linear degradation in the percentage of supported QPS; however, the slope of the curve in *moses* is much more pronounced showing a much higher degradation. On average, both applications consume a similar amount of bandwidth. We looked the reasons behind this behavior and we found that it is mainly due to the fact that the distribution of the average memory bandwidth in *moses* experiences at run-time much higher peaks than in *sphinx*, which shows a more regular pattern (see Fig. 4(c)). That is, the *upper whisker* is much higher (over $10^4 \times 4$) in *moses* than in *sphinx* (around $10^4 \times 4$). Finally, limiting



(a) Supported QPS (b) LLC misses

Fig. 6. Impact of limiting the memory BW on the maximum supported QPS and LLC misses.



(a) Disk BW (b) Supported QPS

Fig. 7. Impact of stressing the disk BW by means of a microbenchmark on the maximum supported QPS and reduced disk BW consumption.

the main memory bandwidth slows down the execution time of the application; consequently, as a side effect, the number of LLC misses also reduces linearly with the limited bandwidth since they take place over longer time. Fig. 6(b) supports this claim. As observed, the obtained values almost match those obtained with the decrease in the supported QPS.

Cloud provider actions for performance improvements. *The memory bandwidth can have a strong impact on the performance of the workloads. The cloud provider should provide memory bandwidth enough to applications (e.g., CPU scalable) suffering a high number of LLC misses, otherwise, the performance can dramatically drop.*

7.4. Disk bandwidth analysis

The disk does not prevent the performance of disk oriented applications from growing in the 8-threaded scenarios since the studied applications have a small consumption compared to the available total bandwidth (see Section 6.3). Despite this fact, the disk bandwidth is a concern in public clouds, especially in those situations where multiple VMs try to perform I/O operations at the same time.

To test this claim, we have used the microbenchmark *stress-ng* to introduce a constant stress on the disk bandwidth by performing random write operations. Different interference levels have been explored by limiting the disk bandwidth assigned to *stress-ng* to 50 MB/s, 100 MB/s and 200 MB/s, which has been done with the libvirt tool. The remaining disk bandwidth (up to 500MB/s) is available for the studied benchmark. We analyzed the impact on the supported QPS and on how the consumed bandwidth reduces over isolated execution.

Fig. 7 shows the results for the three disk applications, *shore*, *xapian* and *moses*. Fig. 7(a) shows the impact of *stress-ng* on the consumed bandwidth of the disk applications for the studied interference levels. The consumed bandwidth significantly reduces over isolated execution (*None*), especially in *shore* and *xapian*. This happens because, their bandwidth consumption keeps similar across all the execution intervals; that is, the standard deviation of the gathered bandwidth values is very small and all the values fall close to the median (see Fig. 4(e)), and *stress-ng* reduces the available bandwidth across all the execution time. In contrast, *moses* consumes most of its bandwidth at the beginning of its execution, thus its consumption is only affected at that execution phase.

Fig. 7(b) shows the impact of the bandwidth reduction on the supported QPS. It is strongly related with the type of disk operation performed. It can be noticed that disk read workloads (i.e., at least 80% of the operations are reads) suffer less performance degradation, even when their bandwidth decreases over 60% in *xapian*. In this case, performance drops by 20% when *stress-ng* consumes 200MB/s. In contrast, in disk write workloads like *shore*, the supported QPS decrease (in percentage) is higher than the bandwidth consumption decrease.

Cloud provider actions for performance improvements. *There is a strong connection between disk bandwidth consumption and the performance of disk applications, especially for write disk workloads that present an homogeneous disk consumption across the execution time. The cloud provider should take especial care with these applications by pinning them to machines with balanced or low disk bandwidth consumption.*

8. Conclusions

Multithreaded latency-critical and streaming applications represent an important subset of cloud workloads as well as streaming applications like *media-streaming*. Understanding how performance is affected by the utilization of the major system resources is a major concern of public cloud providers.

This paper has characterized cloud applications in order to identify key relationships between performance and resource

consumption. The results show that CPU resources can be significantly reduced by taking into account that the performance of some applications does not scale with the number of threads and that threads of Hyper-Threading insensitive applications can be allocated to the same physical cores without affecting performance. Identifying these applications at run-time, however, is challenging. We have shown that this challenge can be successfully dealt with by analyzing the utilization of the major system components. In addition to CPU, we have also studied how the share each application has of other major shared system resources impacts on performance. Experimental results show that some applications can suffer performance losses by more than 80% if not provided with a big-enough share of its critical resource.

The conclusions of the presented studies and the discussions identifying cause–effect relationships among the utilization of different system components can serve as basis for cloud providers to develop virtualization strategies.

CRediT authorship contribution statement

Lucía Pons: Software, Investigation, Data curation, Visualization, Writing – original draft, Writing – review & editing. **Josué Feliu:** Software, Investigation, Writing – original draft, Formal analysis, Writing – review & editing. **José Puche:** Software, Investigation, Writing – original draft. **Chaoyi Huang:** Supervision, Validation. **Salvador Petit:** Conceptualization, Methodology, Formal analysis, Writing – review & editing. **Julio Pons:** Resources, Visualization, Software. **María E. Gómez:** Resources, Methodology, Writing – review & editing. **Julio Sahuquillo:** Conceptualization, Project administration, Funding acquisition, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been supported by Huawei Cloud, and in part by Spanish Ministerio de Universidades under grant FPU18/01948, and by Spanish Ministerio de Universidades and European ERDF under grant RTI2018-098156-B-C51.

Appendix. Effect of hyper-threading in different processor architectures

The fact that SMT processors can improve system throughput with just a little extra hardware cost has made most processor manufacturers (e.g., IBM, Intel or AMD) include SMT processors among their products, especially those deployed in the server segment of the market.

Regarding Intel Hyper-Threading (Intel name to refer to the simultaneous multithreading or SMT paradigm) cores, the performance improvement (e.g., in terms of instructions per cycle or IPC) from running the threads in single-thread (ST) mode with respect to SMT mode slightly differs among different processors. This is specially true for compute intensive applications and mainly happens because the issue ports have experienced minor differences across different processor generations.

To prove this claim, we tested three different Intel processors with different microarchitectures from the last years:

1. Intel Broadwell processor – Intel Xeon CPU E5-2620 v4 CPU (launched Q1 2016).
2. Intel Skylake processor – Intel Xeon Silver 4116 CPU (launched Q3 2017).
3. Intel Cascade Lake processor – Intel Xeon Silver 4210R CPU (launched Q1 2020).

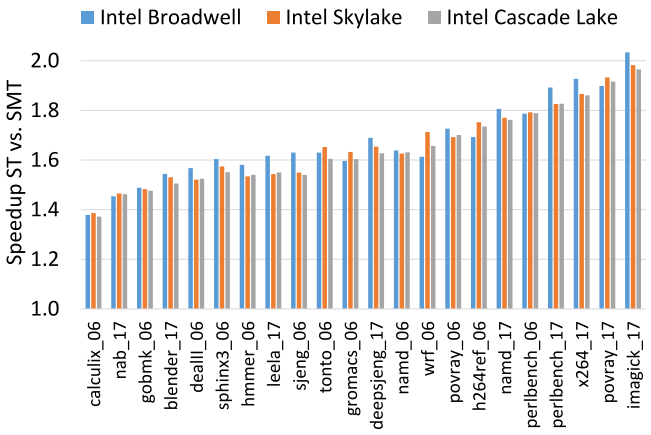


Fig. A.8. Speedup of ST vs. SMT for SPEC CPU applications in three different machines.

The experiment quantifies the speedup (in terms of IPC) of ST execution with respect to SMT execution. For comparison purposes, we have studied the behavior of SPEC CPU benchmarks [42]. Two instances of each benchmark were launched in each run. In ST mode, each instance was pinned to a different physical core, and in SMT mode, both instances where pinned to the same physical core. Fig. A.8 shows the results for speedup obtained for the SPEC CPU applications both from 2006 (*application_06*) and 2017 (*application_17*) when executed in the three processors. For the purposes of quantifying the effect of Hyper-Threading (i.e., CPU resources), memory-intensive applications were omitted since the cache hierarchy widely differs among them. The Intel Broadwell has an inclusive L3 cache of larger capacity (20 MB compared to non-inclusive 16.5 MB in Skylake and 13.8 MB in Cascade Lake) and a smaller L2 cache (256 KB compared to 1 MB both for the Intel Skylake and Cascade Lake processors). The results show that for all the studied applications, similar improvements are obtained, and differences never exceed 5%.

The conclusions of this experiment show that the results of the paper can be extended to other Intel Hyper-Threading processors, at least from the last 6 years as evaluated in this appendix.

References

[1] MongoDB, 2020, Accessed on: Sep. 28, 2020. [Online], Available: https://www.mongodb.com/.

[2] NGINX, 2020, Accessed on: Sep. 28, 2020. [Online], Available: https://nginx.org/.

[3] J. Park, S. Park, W. Baek, Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers, in: Proceedings of the Fourteenth EuroSys Conference 2019, in: EuroSys '19, 2019, pp. 1–16, http://dx.doi.org/10.1145/3302424.3303963.

[4] J. Park, S. Park, M. Han, J. Hyun, W. Baek, Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers, in: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, in: PACT '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 1–14, http://dx.doi.org/10.1145/3243176.3243211.

[5] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, C. Kozyrakis, Heracles: Improving resource efficiency at scale, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, in: ISCA '15, 2015, pp. 450–462, http://dx.doi.org/10.1145/2749469.2749475.

[6] S. Chen, C. Delimitrou, J.F. Martínez, PARTIES: Qos-aware resource partitioning for multiple interactive services, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS '19, 2019, pp. 107–120, http://dx.doi.org/10.1145/3297858.3304005, URL http://doi.acm.org/10.1145/3297858.3304005.
[7] H. Kasture, D. Sanchez, Tailbench: a benchmark suite and evaluation methodology for latency-critical applications, in: 2016 IEEE International Symposium on Workload Characterization (IISWC), 2016, pp. 1–10.
[8] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A.D. Popescu, A. Ailamaki, B. Falsafi, Clearing the clouds: a study of emerging scale-out workloads on modern hardware, in: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, 2012.
[9] Intel Corporation, Intel RDT library, 2019, https://github.com/intel/intel-cmt-cat/tree/master/lib.
[10] R. Cochran, C. Hankendi, A.K. Coskun, S. Reda, Pack amp; Cap: Adaptive DVFS and thread packing under power caps, in: 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2011, pp. 175–185.
[11] R. Schöne, T. Ilsche, M. Bielert, D. Molka, D. Hackenberg, Software controlled clock modulation for energy efficiency optimization on intel processors, in: 2016 4th International Workshop on Energy Efficient Supercomputing (E2SC), 2016, pp. 69–76.
[12] C. Delimitrou, C. Kozyrakis, Quasar: Resource-efficient and qos-aware cluster management, in: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS '14, 49, 2014, pp. 127–144, http://dx.doi.org/10.1145/2541940.2541941.
[13] Linux containers, 2020, Accessed on: Sep. 28, 2020. [Online], Available: https://linuxcontainers.org/.
[14] C. Delimitrou, C. Kozyrakis, Hcloud: Resource-efficient provisioning in shared cloud systems, SIGARCH Comput. Archit. News (2016) 473–488, http://dx.doi.org/10.1145/2980024.2872365.
[15] G. Torres, C. Liu, Adaptive virtual machine management in the cloud: A performance-counter-driven approach, Int. J. Syst. Serv.-Oriented Eng. (2014) 28–43, http://dx.doi.org/10.4018/ijssoe.2014040103.
[16] D. Novaković, N. Vasić, S. Novaković, D. Kostić, R. Bianchini, DeepDive: Transparently identifying and managing performance interference in virtualized environments, in: Proceedings of the 2013 USENIX Conference On Annual Technical Conference, USENIX ATC'13, 2013.
[17] Are noisy neighbors in your data center keeping you up at night?, Tech. rep., Intel Cloud Technology, 2017, URL https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-rdt-infrastructure-paper.pdf.
[18] S. Weil, S. Brandt, E. Miller, D. Long, C. Maltzahn, Ceph: A scalable, high-performance distributed file system, 2006, pp. 307–320,
[19] I. Habib, Virtualization with KVM, Linux J. 2008 (166) (2008).
[20] QEMU, 2021, Accessed: 2021-04-17, https://www.qemu.org/.
[21] Libvirt: The virtualization API, 2021, Accessed: 2021-04-17, https://libvirt.org/.
[22] Open vswitch, 2020, Accessed on: Sep. 28, 2020. [Online], Available: https://www.openvswitch.org/.
[23] DPDK, 2020, Accessed on: Sep. 28, 2020. [Online], Available: https://www.dpdk.org/.
[24] O. Sefraoui, M. Aissaoui, M. Eleuldj, Openstack: Toward an open-source solution for cloud computing, Int. J. Comput. Appl. 55 (2012) 38–42.
[25] Ceph snapshots, 2021, Accessed on: Jan. 8, 2021. [Online], Available: https://docs.ceph.com/en/latest/rbd/rbd-snapshot/.
[26] D. Marr, F. Binns, et al., Hyper-threading technology architecture and microarchitecture, Intel Technol. J. 6 (2002).
[27] D.M. Tullsen, S.J. Eggers, H.M. Levy, Simultaneous multithreading: Maximizing on-chip parallelism, in: Proceedings 22nd Annual International Symposium on Computer Architecture, 1995, pp. 392–403.
[28] J. Feliu, J. Sahuquillo, S. Petit, J. Duato, Addressing fairness in SMT multicores with a progress-aware scheduler, in: 2015 IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 187–196, http://dx.doi.org/10.1109/IPDPS.2015.48.
[29] R. Baeza-Yates, Applications of web query mining, in: European Conference on Information Retrieval, 2005, pp. 7–22.
[30] D.G. Feitelson, Workload Modeling for Computer Systems Performance Evaluation, Cambridge University Press, 2015.
[31] L.A. Barroso, U. Hölzle, The case for energy-proportional computing, Computer 40 (12) (2007) 33–37.
[32] C. Delimitrou, C. Kozyrakis, Amdahl's law for tail latency, Commun. ACM 61 (8) (2018) 65–72, http://dx.doi.org/10.1145/3232559.
[33] D. Menasce, Qos issues in web services, IEEE Internet Comput. 6 (6) (2002) 72–75, http://dx.doi.org/10.1109/MIC.2002.1067740.
[34] F. Raimondi, J. Skene, W. Emmerich, Efficient online monitoring of web-service SLAs, in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, in: SIGSOFT '08/FSE-16, Association for Computing Machinery, New York, NY, USA, 2008, pp. 170–180, http://dx.doi.org/10.1145/1453101.1453125.
[35] W.S. Cleveland, Robust locally weighted regression and smoothing scatterplots, J. Am. Statist. Assoc. 74 (368) (1979) 829–836, http://dx.doi.org/10.1080/01621459.1979.10481038.
[36] J.D. Triveri, LOESS - nonparametric scatterplot smoothing in python, 2018, Accessed: 2020-12-21, http://www.jtrive.com/loess-nonparametric-scatterplot-smoothing-in-python.html.
[37] Y. Mao, E. Kohler, R.T. Morris, Cache craftiness for fast multicore key-value storage, in: Proceedings of the 7th ACM European Conference on Computer Systems, in: EuroSys '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 183–196, http://dx.doi.org/10.1145/2168836.2168855.
[38] S. Tu, W. Zheng, E. Kohler, B. Liskov, S. Madden, Speedy transactions in multicore in-memory databases, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, in: SOSP '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 18–32, http://dx.doi.org/10.1145/2517349.2522713.
[39] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, M.E. Gómez, Application clustering policies to address system fairness with intel's cache allocation technology, in: Procedings of PACT, 2017, pp. 194–205.
[40] L. Pons, V. Selfa, J. Sahuquillo, S. Petit, J. Pons, Improving system turnaround time with intel CAT by identifying LLC critical applications, in: Proceedings of Euro-Par, 2018, pp. 603–615.
[41] L. Pons, J. Sahuquillo, V. Selfa, S. Petit, J. Pons, Phase-aware cache partitioning to target both turnaround time and system performance, IEEE Trans. Parallel Distrib. Syst. 31 (11) (2020) 2556–2568, http://dx.doi.org/10.1109/TPDS.2020.2996031.
[42] A. Navarro Torres, J. Alastruey Beneda, I. Marín, V. Yúfera, Memory hierarchy characterization of SPEC CPU2006 and SPEC CPU2017 on the intel xeon skylake-SP, PLOS ONE (2019) http://dx.doi.org/10.1371/journal.pone.0220135.

Lucia Pons received the B.S. and M.S. degrees in computer engineering from the Universitat Politècnica de València (UPV), Spain, in 2018 and 2019, respectively. She is currently working toward a Ph.D. degree at the Department of Computer Engineering (DISCA) of the same university. Her Ph.D. research focuses on cache partitioning approaches and efficient use of shared resources in multi-core.

Josué Feliu received his M.Sc. and Ph.D. degrees in computer engineering from the UPV, Spain, in 2012 and 2017, respectively. He is currently working as a postdoctoral researcher at the Universidad de Murcia. His research interests include scheduling strategies and performance modeling for multicore and multithreaded processors. He was awarded the "IEEE TCSC Outstanding Ph.D Dissertation Award" in 2017.

José Puche received his B.S. from UCLM and his M.S. in Computer Engineering from the UPV, Spain in 2014 and 2015, respectively. He is currently a Ph.D. student at the Parallel Architecture Group (GAP) of the Universitat Politècnica de València. His research interests include processor and memory hierarchy architecture.

Chaoyi Huang received his B.S. degree in mechanical engineering, M.S. degree in computer aided quality management from Huazhong University of Science & Technology, China. He is an expert in Cloud BU, Huawei Technologies Co., Ltd. His current research interests is improving cloud resource efficiency, including technologies like inter-VM interference detection and control, dynamic VM resizing.

Salvador Petit received the Ph.D. degree in computer engineering from the UPV. Since 2009, he has been an Associate Professor with the DISCA department, where he has taught several courses on computer organization. He has authored over 100 refereed conference and journal papers. His current research interests include multi-core processors, memory hierarchy design, GPU architecture, and resource management.

Julio Pons received the B.S., M.S., and Ph.D. degrees from the UPV, Spain, all in computer engineering. He is an Associate Professor with the DISCA department. He has taught several courses on computer organization and operating systems. His current research interests include multi-core processors, memory hierarchy design, cache sharing and cloud computing.

María E. Gómez received his B.S., M.S., and Ph.D. degrees in Computer Engineering from the UPV, Spain, in 1996 and 2000, respectively. She joined the Department of Computer Engineering (DISCA) at Universitat Politècnica de València in 1996 where she is currently a Full Professor. She has published more than 80 conference and journal papers. She has served on program committees for several major conferences. Her research interests are on processor architecture and interconnection networks.

Julio Sahuquillo received the B.S., M.S., and Ph.D. degrees from the UPV, Spain, all in computer engineering. He is a Full Professor with the DISCA department at the UPV. He has taught several courses on computer organization and architecture. He has authored over 150 refereed conference and journal papers. His current research interests include multicore processors, memory hierarchy design, GPU architecture, and resource management.