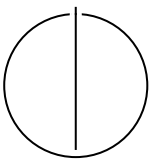
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Analysis of the Noisy Neighbor Problem in AWS

Youssef Jemal



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

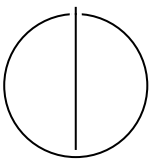
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Analysis of the Noisy Neighbor Problem in AWS

Analyse der "Noisy Neighbor" Problem in AWS

Author:	Youssef Jemal
Examiner:	Prof. Dr. Viktor Leis
Supervisor:	M.Sc. Till Steinert
Submission Date:	22/08/2025



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 22/08/2025 Youssef Jemal

Abstract

Contents

1 Introduction

Infrastructure-as-a-Service (IaaS) is a cloud computing model that provides customers with access to computing resources such as servers, networking, and virtualization. Cloud vendors in general and Amazon Web Services (AWS) in particular abstract the physical placement of virtual machines, providing users with limited transparency about how many tenants are sharing the underlying hardware. This information can be crucial as this co-residency can result in significant performance degradation across different resources such as CPU, memory, network, and storage I/O [12] [18].
In 2017, AWS launched the Nitro system [6], which enhances virtualization in their data centers by offloading most virtualization tasks to dedicated hardware. However, contention can still occur due to other factors such as Simultaneous Multi-Threading (SMT), Network Interface Card queuing, and other system-level bottlenecks.
This thesis provides empirical evidence of the maximum possible performance degradation on AWS and investigates its primary causes. Understanding these performance impacts is important for customers when selecting a suitable instance type for their workloads, as co-located performance degradation may cause them to receive less value than paid for. Throughout this thesis, we leverage different benchmarking tools to assess the extent of resource contention. We ran identical resource-intensive benchmarks in parallel across VMs residing on the same dedicated host. We primarily utilized 5th and 6th general-purpose AWS EC2 instances that leverage the AWS Nitro system. We assess CPU performance degradation across various vendors, namely Intel, Graviton, and AMD, and analyze its contributing factors. Furthermore, we investigate network resource contention, which is important since AWS does not provide exact specifications like other resources, such as CPU and memory. We quantify the degradation that can occur both on throughput and latency across co-located EC2 instances.
The thesis is structured as follows: We start by discussing related work in Section 2. Section 3 introduces the key concepts required to understand this work. Section 4 provides information about our testing infrastructure. Section 5 analyzes CPU contention across 5th and 6th EC2 generations and identifies its causes. In Section 6, we examine network I/O contention and explore its manifestation across throughput and latency, before we conclude in section 7.

2 Related Work

Han et. al [12] investigated public cloud resource contention. They executed CPU, disk, and network I/O benchmarks across up to 48 VMs sharing the same dedicated host. The tests were executed mainly on 3rd (c3), 4th (c4), and 5th (m5d) generation VMs. The results showed considerable performance degradation, with CPU degradation reaching 48% and throughput degradation up to 94%. Throughput degradation was measured in relation to the initial bandwidth available to the VM, i.e., burst bandwidth and not the baseline bandwidth. The paper also analyzed the unexpected CPU performance degradation caused by adding idle Linux VMs on the dedicated host. The measurements were leveraged to train multiple linear regression and random forest models to predict VM co-residency. The linear regression model achieved an R^2 of .942. This could be very practical, as it enables users to relocate their VMs to have access to better performance with less contention.
Rehman et. al [21] analyzed the problem of provisioning variation in public clouds. By running benchmarks and sample MapReduce workloads, they found that provisioning variation can impact the performance by a factor of 5. They argue that this is primarily due to network I/O contention.
Lloyd et. al [18] reported a 25% performance degradation when running compute-intensive scientific modelling web services on pools consisting of m1.large VMs with high resource contention. They developed an approach called Noisy-Neighbor-Detect that leverages the cpuSteal metric to identify VMs with noisy neighbors from a pool of worker VMs. cpuSteal refers to the percentage of time a virtual CPU spends waiting for the hypervisor to allocate a physical CPU to run on.
Many other techniques were developed by researchers to identify or predict resource contention. Govindan et. al [10] developed a software solution called Cuanta that predicts performance interference due to shared chip-level resources, namely cache space and memory bandwidth. Although the performance degradation of consolidated application can be empirically investigated, the number of possible workload placements is combinatorial. A cloud provider hosting M VMs with N VMs per server needs to perform $\frac{M!}{N!\,(M-N)!}$ measurements, which is highly impractical. Cuanta does not require any changes to the hosting platform's software and the prediction complexity is linear to the number of cores sharing the Last Level Cache (LLC), making it a far better alternative than its empirical counterpart. The software provided promising results

with up to 96% accuracy on Intel Core-2-Duo processors.
Some efforts have looked at side channels as a way to detect VM co-location. Side channels are an indirect way of extracting information from a system that designers never intended to expose its implementation details. Inci et. al [14] developed three methods to detect co-located VMs. The first two approaches leveraged Last Level Cache: Cooperative LLC covert channel and Cache profiling. While the former requires cooperation of the victim VMs, the latter operates independently. In the second method, the attacker fills the cache with its own data and after a short pause re-accesses the same buffer while monitoring the memory access time. Low eviction rates indicate that the attacker is likely alone on the host, while high eviction rates point toward VM co-location. The third method is memory bus locking. The idea is for the attacker to launch special instructions that block the memory bus and then analyze the resulting delays to infer VM-colocation. All three methods had a high accuracy in detecting co-location in real commercial cloud settings.

3 Background

3.1 Virtualization

Virtualization is a technology that allows the creation of isolated virtual environments also known as Virtual Machines that run on the same physical server [20]. Each VM has its own operating system and acts as an independent physical computer. The VMs are called "guests" and the physical server is called "host". Virtualization is crucial for the IaaS model offered by cloud providers, as it provides various advantages [20]. It improves resource and cost efficiency by dividing the physical server into multiple isolated instances, each tailored to different workload needs. This reduces the amount of unused capacity that occurs when a server is dedicated to just one task.
The main component that handles the necessary tasks for virtualization is the Virtual Machine Monitor (VMM) also called hypervisor [6]. Most of the instructions that are executed by the virtual machines run natively on the CPU and do not require intervention from the VMM, such as arithmetic operations. However, there is a class of privileged instructions that guests can not directly execute on the CPU, such as I/O operations. When such an instruction is encountered, the CPU raises a trap, which signals to the VMM to intervene and emulate the behavior of the instruction [6]. After the emulation is finished, the control is then given back to the guest OS, which is unaware of the underlying emulation. Several optimization techniques have been introduced to reduce virtualization overhead, which will be briefly outlined.
Virtualization cannot be carried out by the VMM alone, as it does not virtualize hardware and therefore can not grant the guests access to the underlying hardware devices such as network interfaces, storage drives, and input peripherals. Device models are required for this [6]. They are basically software components that communicate with the shared hardware and expose multiple virtual device interfaces to the VMs. Device models, along with other management software, run in a special privileged virtual machine called management domain, which represents the host's operating system and has access to all the underlying hardware. This domain is called domain zero or dom0 in the Xen project, and root/parent partition in the Hyper-V project [6]. Since the device models are software-based, they compete for for CPU and other system resources along with the existing VMs and can negatively affect the performance of the guests. The following figure summarizes the architecture of a traditional virtualization

system.

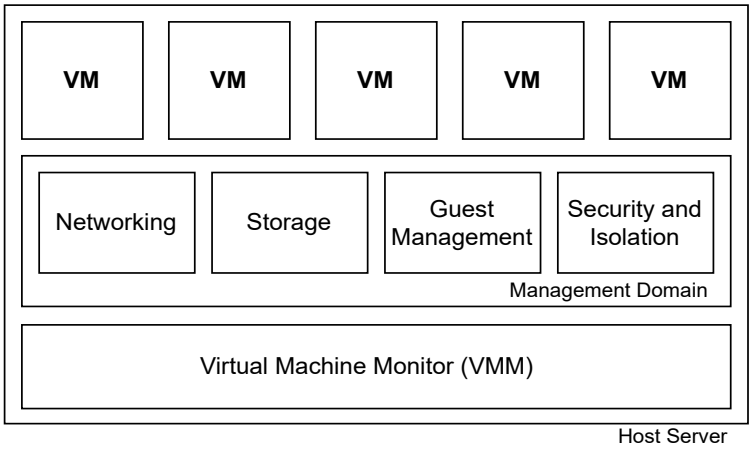


Figure 3.1: Architecture of traditional virtualization Solution

3.1.1 Evolution of Virtualization Solutions

Virtualization technology has evolved significantly over the last decades. It started with full software virtualization, where the guest OS is unmodified and "unaware" of the virtual environment. Privileged instructions are trapped by the CPU, and the hypervisor emulates the sensitive instructions using binary translation [11]. This is, however, very slow and can make the host applications run 2x to 10x slower [11].
Then, paravirtualization was introduced, where the guest OS is modified to interact directly with the hypervisor via "hypercalls", removing the abstract emulation layer. The downside of this approach is that it introduces additional complexity, as it requires modifications to the guest operating system [13].
The next major leap was hardware-assisted virtualization (HVM), which introduced virtualization support directly on the hardware level by providing highly efficient and fast virtualization commands. This provides a significant improvement in comparison to the previous virtualization techniques, as it reduces the involvement of the host system in handling privilege and address translation space tasks [13]. Intel offers this under the Intel VT-x technology that provides virtualization of CPU and memory. Another important example is Single Root Virtualization (SR-IOV) [6], which is a technology that allows physical PCI devices, such as Network Interface Card (NIC) to expose multiple virtual devices to the hypervisor. The hypervisor can then provide the different virtual machines with direct hardware access to these virtual devices, which significantly increases the I/O performance.

3.1.2 The AWS Nitro System

The Nitro System is a result of a multi-year incremental process of AWS re-imagining the virtualization technology in order to optimize it specifically for their EC2 data centers [6]. The main idea was to decompose the software components, i.e., the device models, that run on the management domain and offload them to independent purpose-built server components. This helps minimize the resource usage caused by software running in the management domain, effectively allowing a near "bare-metal" performance. Figure 3.2 depicts the new AWS architecture for virtualization.

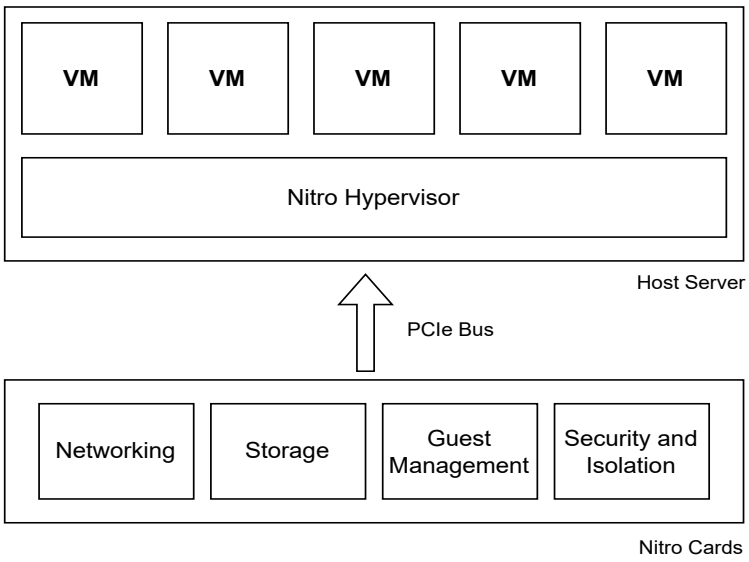


Figure 3.2: Architecture of the AWS Nitro System [6]

There are three main components in the AWS Nitro System.

The Nitro Cards

Nitro cards are dedicated hardware components that operate independently from the EC2's server main board (CPU and memory) and are physically attached to it via PCIe. They "implement all the outward-facing control interfaces used by the EC2 service" responsible for provisioning and managing compute, memory, and storage [6]. They provide all I/O Interfaces as well, such as the ones for storage and networking. These cards employ the previously explained SR-IOV technology to provide direct hardware interfaces to the VMs. Examples of Nitro cards are Nitro cards for I/O and Nitro Controller, which provides the hardware root of trust of the Nitro System.

The Nitro Security Chip

The Nitro Security Chip extends the hardware root of trust and control over the system's main board. It is managed by the Nitro Controller and plays a crucial role in enabling AWS to offer bare-metal instances. Bare-metal instances provide direct access to the physical CPUs and memory of the physical server. They are useful mainly for licensing-restricted business-critical applications, or for specific workloads that require direct access to the underlying infrastructure.
In virtualized environments, the hypervisor is responsible for securing the host's hardware assets. However, in bare metal modes, when no hypervisor is present, the Nitro Security Chip assumes this role and ensures the security of the system firmware from tampering attempts through the system CPUs [6].

The Nitro Hypervisor

The third component is the AWS Nitro Hypervisor, which has significantly fewer responsibilities than traditional hypervisors, as most virtualization tasks are offloaded to the Nitro cards. It has three main functions. It is responsible for partitioning memory and CPU by using the virtualization commands provided by the processor. It is also in charge of assigning the virtual hardware interfaces provided by the Nitro cards to the Virtual Machines and handling the machine management commands that come from the Nitro Controller (start, terminate, stop, etc.) [6].

3.2 Simultaneous Multi-threading

Before we dive into SMT, it is important to understand which problem it tries to solve and what the motivation behind it is.
A processor consists of a few hundred registers, load/store units, and a couple of multiple arithmetic units. The main goal is to keep all these resources as busy as possible. Multiple techniques have been employed to achieve this, such as instruction pipelining, superscalar architecture, and out-of-order execution [25]. Pipelining is a technique that breaks down the execution of an instruction into several distinct stages, with each stage using separate hardware resources [27]. During each CPU cycle, instructions advance from one stage to another. This allows the CPU to work on multiple instructions simultaneously, each being on a different stage. In a perfect scenario, where all instructions are independent, the processor can work simultaneously on n instructions, with n being the depth of the pipeline, i.e., the number of stages. The following table depicts a simple example of a five-stage pipeline. At the fifth clock cycle, the CPU is simultaneously working on five instructions.

Instr. No. \ Clock Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

Figure 3.3: Basic five-stage pipeline (IF = Instruction fetch, ID = Instruction decode, EX = execute MEM = memory read, WB = Write back to memory)

Modern processors are also superscalar. This means that each processor can start executing more than one instruction per cycle by dispatching them to different execution units [25]. Issue width is an important characteristic of modern CPUs, and it represents the maximum number of instructions that can be started in a single clock cycle. Although these optimizations significantly increase the processor throughput, the dependency between the instructions and the long latency operations of the executing threads limits the usage of the available execution resources [25]. Out-Of-Order execution partially solves this problem, but it is still not enough, as it still dispatches instructions from the same thread, where the dependency between the instructions is inherently high. The wastages that occur on the processor can be categorized into two categories: Horizontal and vertical waste [25]. Horizontal waste occurs when the CPU is not able to fully saturate the issue width of the processor. Vertical waste occurs when the processor is not able to start any instruction at all on a given cycle because of the dependency to the executing instructions or delays such as memory latency. Traditional multithreading addresses this issue by switching to a different thread whenever the currently executing one stalls. This approach, however, only mitigates vertical waste, as it still issues instructions from only one thread at any given cycle [25].

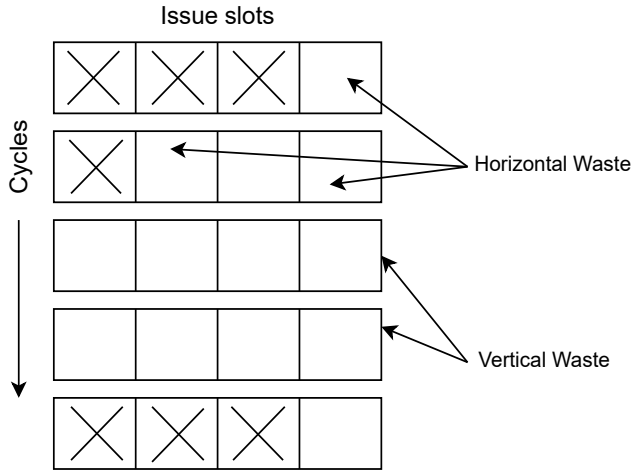


Figure 3.4: Vertical waste vs. horizontal waste

This is where Simultaneous Multi-Threading comes into play. SMT is a technique that helps enhance the overall efficiency of superscalar CPUs by improving the parallelization of computation [27]. This technology allows the physical core to dispatch instructions from more than one thread per cycle without requiring a context switch [25], effectively transforming each physical core into two (or more) "logical" cores. The idea is that instructions from different threads provide greater independence, which results in a better utilization of the core's execution resources [27]. To achieve this, some processor resources are duplicated, e.g., those that store the architectural state, such as registers and program counters [27]. However, the logical cores still share the same execution resources, which can create contention, especially if both threads have the same workload nature, e.g., both are float heavy [19].

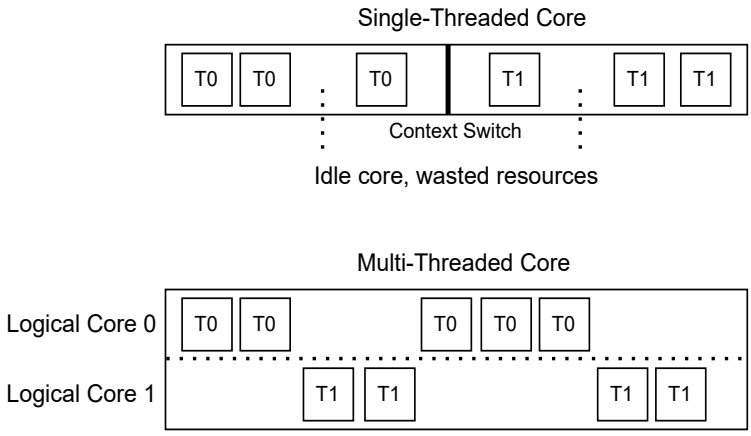


Figure 3.5: Single-Threaded Core vs. Multi-Threaded Core

Both Intel and AMD implement this technology in their modern CPUs, providing two threads per physical core. Intel brands it as Hyper-Threading, while AMD uses the standard term SMT. In the AWS dedicated hosts that run on an Intel or AMD CPU with hyperthreading enabled, the number of vCPUs is always equal to the double of the number of physical cores, with each vCPU corresponding to a hyperthread. This, however, opens up the possibility for CPU contention, if two virtual machines have access to vCPUs that share the same underlying physical core. Unlike Intel and AMD CPUs, AWS-designed Graviton processors, built around the ARM architecture, do not support hyper-threading and expose one execution context, i.e., vCPU for each physical core [23]. This allows for a better isolation between the different tenants as there is no resource sharing between the different vCPUs apart from the last level cache and the memory system [23].

3.3 Dynamic Voltage and Frequency Scaling (DVFS)

DVFS is a technique that allows the CPU to adjust its frequency based on the workload. When executing computationally intensive tasks, it allows the processor to jump to a higher frequency. Conversely, for light workloads or when the processor is idle, a lower clock rate is adopted. The goal is to create a balance between performance and energy efficiency.
Intel implements this technology under the name Intel® Turbo Boost Technology [15]. Its latest version allows frequency scaling on a per-core basis. Modern AMD processors also support similar per-core granularity through Precision Boost 2 [4]. Frequency increases are, however, subject to certain conditions: The processor ensures that pre-

defined temperature, as well as power consumption limits are not exceeded. In cloud environments, this behavior can potentially lead to performance degradation when the processor is forced to lower a core's frequency from its peak level due to increasing power usage and processor temperature, caused by busy neighbors.

4 Testing Infrastructure

All tests were performed in the AWS us-east-2 Ohio region. The EC2 instances ran Ubuntu Server 24.04 LTS Linux. Throughout the experiments, all the VMs were provisioned on dedicated hosts that are in the same availability zone and Virtual Private Cloud. This is particularly important for network I/O experiments, as the traffic between VMs sharing the same AZ and VPC is free of charge. We ran parallel benchmarks on general-purpose and compute-intensive dedicated hosts from the 5th, 6th, and 7th generations. We used dedicated hosts to ensure all nodes share the same physical server. This provides a controlled environment that allows the measurement of degradation caused by resource contention, without any external interference.
To deploy the resources for the different experiments, we used Terraform, which is an Infrastructure-as-Code (IaC) tool developed by HashiCorp. It allows defining and provisioning resources using the HashiCorp Configuration Language (HCL), ensuring the automation and reproducibility of the benchmarks. Additionally, we used Distexprunner [28], which is a tool written in Python that facilitates writing and executing bash commands remotely across multiple nodes via their public IPs. Our experiments generated JSON or CSV files that were gathered in an S3 bucket using Distexprunner. We also implemented multiple scripts in Python using mainly the re package [8] for parsing raw data and csv [7] for working with comma-separated values. These scripts were essential to transform the raw data generated by our experiments into clean data that can be used for visualization and analysis.
We prepared different Amazon Machine Images (AMI) that have the required software pre-installed (Python, Distexprunner, benchmarking tools, and monitoring utilities). This minimizes the boot time of our instances and prevents configuration drift between runs.
These conditions apply to both the network and CPU experiments. The specific experimental setups will be discussed in detail within the methodology of each main section.

5 CPU Contention

5.1 Methodology

To generate CPU stress, we used sysbench [17], a powerful cross-architecture tool that can be used for Linux performance benchmarks. As a CPU stress-testing tool, it deterministically checks all prime numbers up to 10,000 (default value) by dividing each candidate number by all integers from 2 up to its square root [9]. The number of worker threads and the aggregated workload of the created threads can be specified as arguments. The total execution runtime is then used as a comparison metric between the different experiments. For comparison purposes, we also developed our CPU stressing tool called *cpu_burn* written in C language. The program takes two arguments, the first being the number of operations that each created thread will perform, and the second representing the number of worker threads. It outputs the total wall-clock runtime that was needed for the execution of the job. We compiled the program using GCC with optimization level of O0, to ensure no compiler optimizations altered the program's behavior. Each thread executes the function defined under `perform_work()`. As stated previously, The argument `work->operations` is specified by the user.

```
void* perform_work(void* arg) {
    ThreadWork* work = (ThreadWork*)arg;
    double x = 0.0;
    for (long long i = 0; i < work->operations; ++i) {
        x += i * 0.000001;
    }
    work->result = x;
    return NULL;
}
```

Listing 5.1: Workload of the *cpu_burn* tool

This section investigates CPU contention on general-purpose instances from 5th, 6th, and 7th EC2 generations. We analyzed the potential degradation on hosts that support Simultaneous Multi-Threading: namely m5, m6i, m6a hosts. We then investigated

contention on hosts running on single-threaded processors, particularly the m6g and m7a hosts. Key Performance Indicators are described in Table 5.1. We notice that the first three dedicated hosts have a number of vCPUs twice the number of the physical cores, as these hosts support SMT with two threads per physical core. Hosts with a clock speed of "Up to X" support Frequency Scaling. The m6g instance has the best price per vCPU ratio, although each vCPU is mapped to a physical and not to a logical core.

KPI	m5	m6i	m6a	m7a	m6g
Processor [26]	Intel Xeon Platinum 8175/ Intel Xeon Platinum 8280	Intel Xeon 8375C	AMD EPYC 7R13 Processor	AMD EPYC 9R14 Processor	AWS Graviton2
Hypervisor [2]	Nitro v2	Nitro v4	Nitro v4	Nitro v4	Nitro v2
vCPUs [22]	96	128	192	192	64
Physical CPUs [22]	48	64	96	192	64
Clock speed (GHz) [5]	Up to 3.1	Up to 3.5	Up to 3.6	Up to 3.7	2.5
price/hour [22]	$5.069	$6.758	$9.124	$12.24	$2.71
price/vCPU/hour	$0.053	$0.053	$0.048	$0.063	$0.042

Table 5.1: KPIs for AWS Dedicated Host families

The experiment is structured as follows: A node, referred to as test node is first deployed on the dedicated host. Next, neighbors that fully utilize their vCPUs are incrementally added. We analyze the effect of adding busy neighbors on the runtime of running cpu benchmarks on the test node. Figure 5.1 provides a visualization of the experiment.

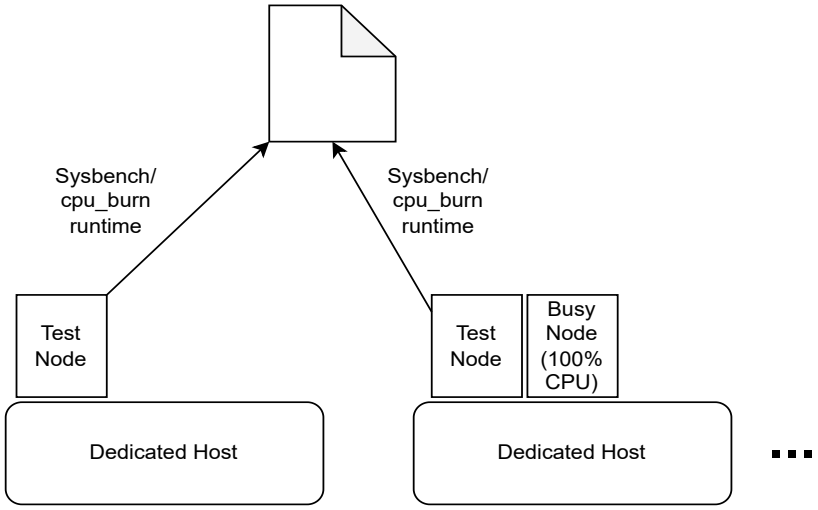


Figure 5.1: CPU contention experiment

The experiment was repeated but with idle neighbors instead of busy. To identify the cause of the observed degradation, we also natively ran the benchmarking tools on the metal instance of each family. Metal instances run directly on the physical host without the need for the hypervisor. They provide users with access to all the physical CPUs.

5.2 Hypotheses

The following hypotheses guide our investigation of CPU performance degradation:

H1: Overhead introduced by virtualization, such as hypervisor overhead, is expected to be minimal and practically insignificant. The substantial improvements introduced by hardware-assisted virtualization and the Nitro System support this assumption.

H2: For hosts that support SMT, physical core co-location between the co-tenants will likely create measurable performance degradation. If the vCPUs of the test node are mapped to several physical cores, we should witness a gradual performance loss, as busy neighbors start sharing these cores with the test node.

H3: Frequency scaling can be a reason for performance degradation. When the test node runs alone on the dedicated host, it should operate at peak frequency and achieve optimal performance. As busy neighbors are added, the processor's average frequency is likely to decrease due to power and temperature limits, leading to performance loss.

5.3 Contention under SMT

5.3.1 m5 Family

The first set of experiments will be conducted on an m5 dedicated host. This host features either the 1st or 2nd generation Intel Xeon Platinum 8000 Series processor, namely Skylake-SP or Cascade Lake [3]. The following table provides an overview of the different instance types belonging to this family.

Instance Size	vCPU	Memory (GiB)
m5.large	2	8
m5.xlarge	4	16
m5.2xlarge	8	32
m5.4xlarge	16	64
m5.8xlarge	32	128
m5.12xlarge	48	192

Table 5.2: m5 instance specifications [3]

The m5 dedicated host used the 2nd-generation Intel CPU in all our benchmarks. We repeated the experiments using hosts running on the 1st-generation CPU, and we received the same behavior, which excludes variation caused by hardware heterogeneity. For our first experiment, we used 2xlarge instances, each featuring 8vCPUs and 32 GiB RAM. The maximum number of nodes is 12. The results can be seen in Figure 5.2.

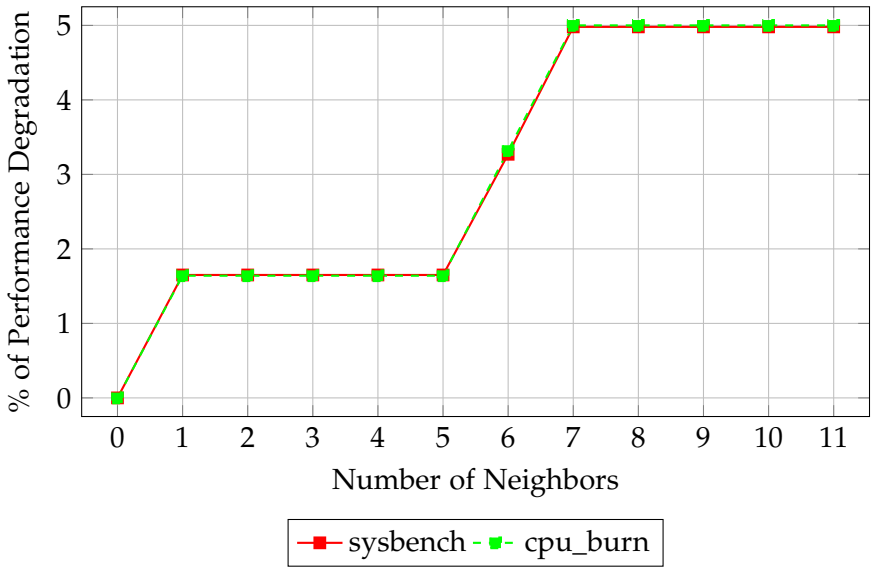


Figure 5.2: Effect of adding busy neighbors on the CPU speed of the test node (m5.2xlarge)

We notice a very similar degradation pattern for the two tools we used. Adding the first neighbor added a performance degradation of 1.6% on our test node. The runtime then remained constant for the next four neighbors. Afterwards, the sixth neighbor degraded the performance further to 3.3%. The seventh neighbor introduced the last witnessed decrease in the performance to reach 5% in both experiments.
This experiment alone does not allow us to pinpoint the reason behind the performance degradation. We repeat the same experiment but adding idle VMs. Figure 5.3 summarizes the results.

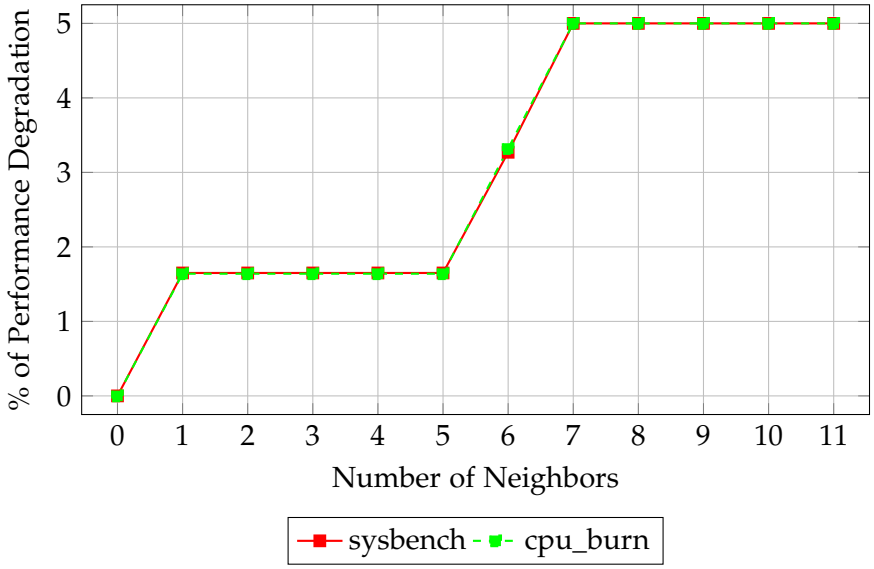


Figure 5.3: Effect of adding idle neighbors on the CPU speed of the test node (m5.2xlarge)

We notice the exact same degradation pattern of the earlier experiment. This result is unexpected and undermines the hypothesis that the performance degradation is due to physical core co-location between the different tenants as we would have expected the effect to be less pronounced when adding idle VMs.
To explore this further, we repeat the experiment using m5.large instances, of which the dedicated host can provision 48. The results of our experiment can be seen in Figure 5.4. In this case as well, adding busy or idle neighbors provided the exact same results.

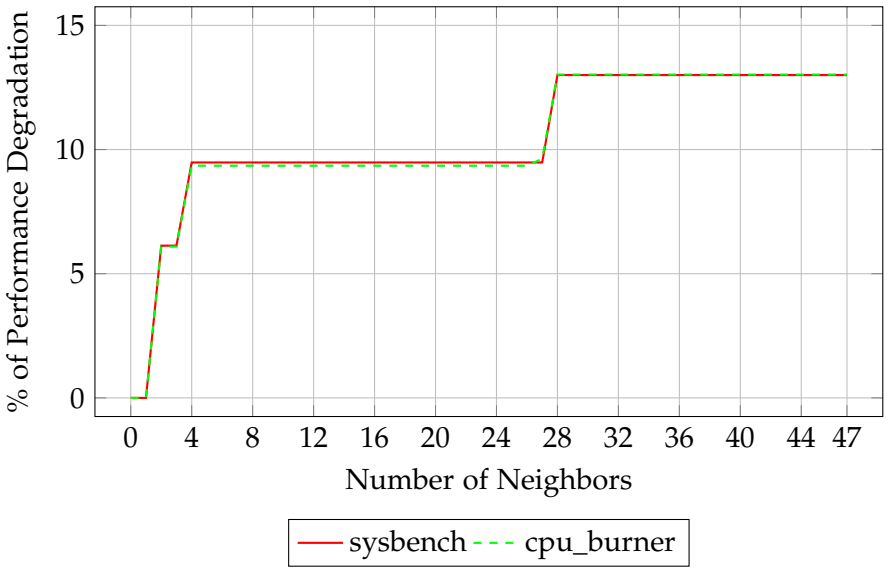


Figure 5.4: Effect of adding busy/idle neighbors on the CPU speed of the sysbench and cpu_burn command on the test node using m5.large instances

We observe the same performance degradation pattern between the two tools. The second neighbor has introduced the first performance degradation of roughly 6%. The 4th neighbor increased this degradation to 9,5%. The runtime then remained constant for the next 23 neighbor, as they had no effect on our test node. The 28th neighbor then introduced the last performance degradation reaching the maximum performance degradation of 13% with both tools.
To have a full picture of the performance degradation across the different instance types, we repeated the experiment using the remaining instance types and summarized the results in table 5.3. The results are always similar between the two tools. From this point onward, we'll proceed exclusively with the *cpu_burn* tool. Adding idle or busy neighbors constantly provided the same results. At each level, we repeated the *cpu_burn* command 10 times and then considered the average of these 10 values.

Instance type	large	xlarge	2xlarge	4xlarge	12xlarge
Maximum Nodes	48	24	12	6	2
Degradation (Busy/Idle) %	13	13	4.8	3.25	0

Table 5.3: Maximum achievable performance degradation on our test node across various m5 instance types

The biggest performance degradation happens when using large and xlarge instances with almost the same percentage of 13%. It then drops to 5% for the 2xlarge type, as seen in figure 5.2. We notice a further decrease in the performance degradation for the 4xlarge type to 3.25% and then its complete absence when using the 12xlarge type, of which the dedicated host can provision 2.
To investigate this issue further we run the experiment directly on the bare-metal m5.instance. For this, we use the *cpu_burn* tool and incrementally increase the number of threads that are created and examine whether we witness any performance degradation. Figure 5.5 visualizes the outcome of the experiment.

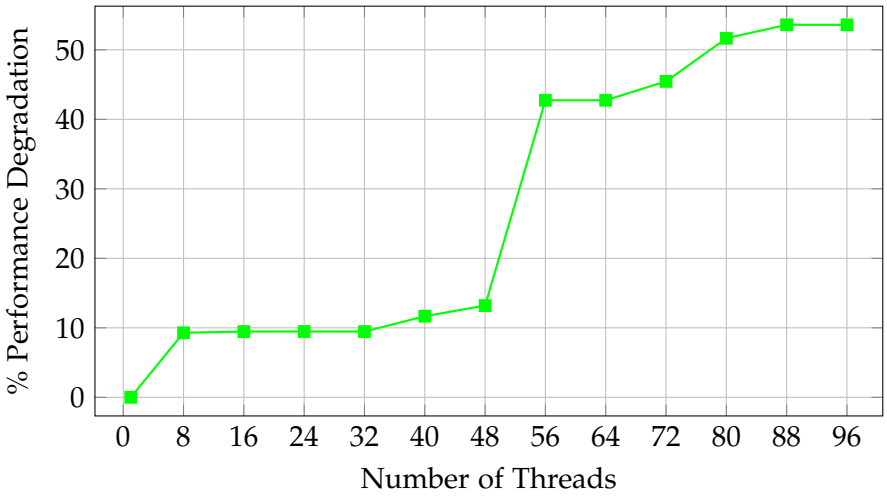


Figure 5.5: Runtime impact of incrementally increasing threads in the *cpu_burn* command on an m5.metal instance

Although the m5.metal has 96 vCPUs i.e., logical cores, we notice a very important pattern of performance degradation throughout the first 96 threads reaching 53%. This is mainly caused by the physical core co-location that happens between the different threads, resulting in resource contention, as the execution resources are not duplicated. In all our previous experiments, the instances initially started with a nominal baseline performance significantly worse than running the exact number of threads directly on the metal instance. Figure 5.6 compares the runtime of *cpu_burn* on the test node (all threads are busy) as we keep adding fully busy neighbors, in comparison to running the same number of busy threads directly on the m5.metal.

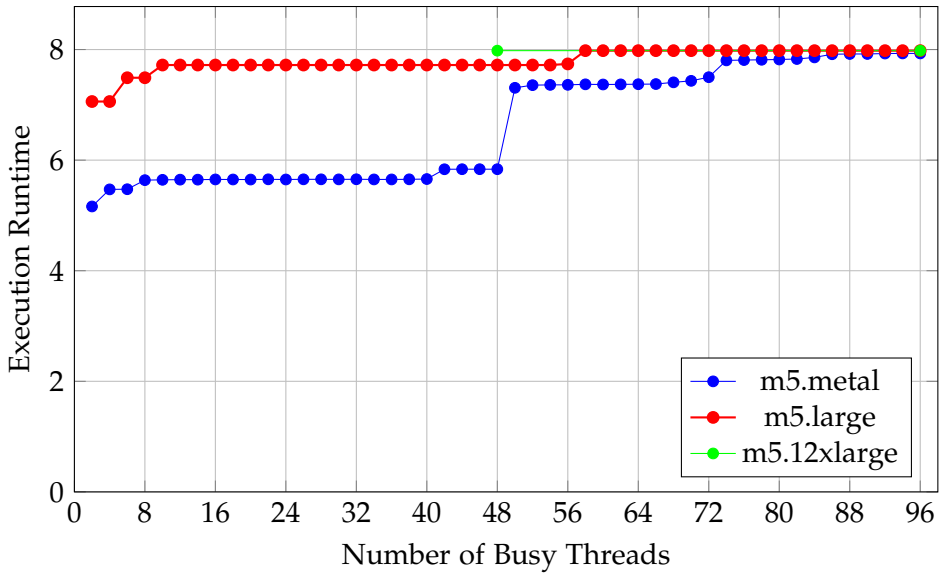


Figure 5.6: Performance of the test node (while incrementally adding busy neighbors) in comparison to running the threads natively on m5.metal

The first two threads finished the execution in 5.165s on the bare metal instance. However, the first m5.large instance that was deployed on the dedicated host took 7.06s to finish, even though it introduced the first two busy threads on the dedicated host and no other VMs exists. This is highly unexpected as we would have expected to see a runtime closer to 5.165s that confirms the promises of AWS of a near bare-metal performance. Instead we notice a difference of almost 36.6%. The same is true for the first m5.12xlarge instance where we witness a difference of 36.7%. Furthermore, we notice that both plots converge almost towards the same value at the maximum number of threads. This strongly undermines the hypothesis that the degradation is due to hypervisor overhead as we would expect to see an even bigger gap (in relation to bare-metal) as more VMs are deployed on the dedicated host. The results for the performance degradation we saw in the previous experiments (Table 2) can be misleading. For bigger instances, we saw a relatively small degradation compared to the smaller instances (large and xlarge). The reason is that the first 12xlarge instance started with a baseline performance that's 13% worse than the performance of the first m5.large instance as Figure 5.6 shows.

The poor initial performance of the first m5.large instance (test node) suggests that the hypervisor pinned its vCPUs to the same physical core, not taking advantage of other non-occupied physical cores. We assume that this allocation technique aims to avoid contention between the different tenants/customers and isolate the vCPUs of

each virtual machine by allocating each pair to the same physical core. This could be advantageous when the customer rents a unique VM, as it is independent of the neighbors' workloads.
To confirm our supposition, we run the *cpu_burn* tool in a m5.2xlarge instance, and incrementally increase the number of busy threads. The results can be seen in the following figure.

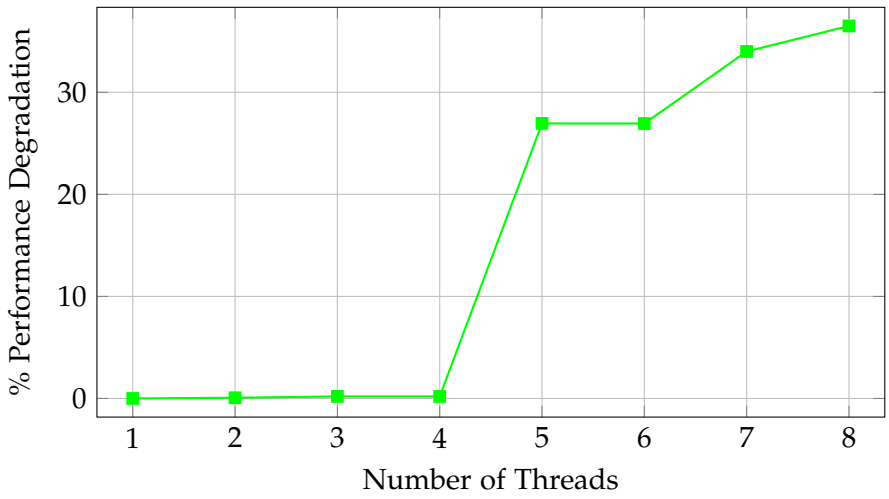


Figure 5.7: Effect of adding busy threads on the CPU performance of a m5.2xlarge instance using *cpu_burn*

We can notice that the biggest degradation happens when adding the 5th thread, which strongly indicates that the instance has access to only 4 physical cores. The hypervisor starts by initially pinning the first 4 threads to an idle physical core to maximize performance but then the 5th thread is allocated to one of these physical cores, sharing the execution resources with another busy thread resulting in a performance degradation of 27%. With this finding, H2 is rejected since co-tenants can not share physical cores.

5.3.2 m6i Family

The m6i family runs on the 3rd Generation Intel Xeon Scalable processor. We investigated the maximum performance degradation that can happen on the test nodes from adding idle and busy VMs with different instance types. Table 5.4 summarizes the results.

Instance type	large	xlarge	2xlarge	4xlarge
Maximum Nodes	64	32	16	8
Degradation (Busy) %	1.48	1.6	1.74	2.3
Degradation (Idle) %	0.05	0.06	0.06	0.07

Table 5.4: Maximum achievable performance degradation on our test node across various m6i instance types

The difference from the previous experiments with the m5 family is that we notice a difference between adding idle or busy neighbors. Adding idle neighbors always results in a sub 0.1% performance degradation which is practically insignificant. This difference between busy and idle instances can't be assigned to hypervisor overhead alone as we notice a sub 0.1% overhead when adding busy instances on the m6g dedicated host that uses the same Nitro v2 Hypervisor, as Table 5.8 shows.
In these experiments, all the instance types started from nearly the same nominal performance level, which is roughly 2% away from the "worst" runtime possible on the metal instance (128 busy threads). This explains the small levels of performance degradation we witnessed in comparison to the m5 family where the m5.large and m5.xlarge instances started with a relatively better (nominal) performance than the other types resulting in a bigger performance degradation compared to them. We now analyze the execution runtime of busy threads directly on the m6i.metal instance, in comparison to to the test node while adding busy 4xlarge neighbors.

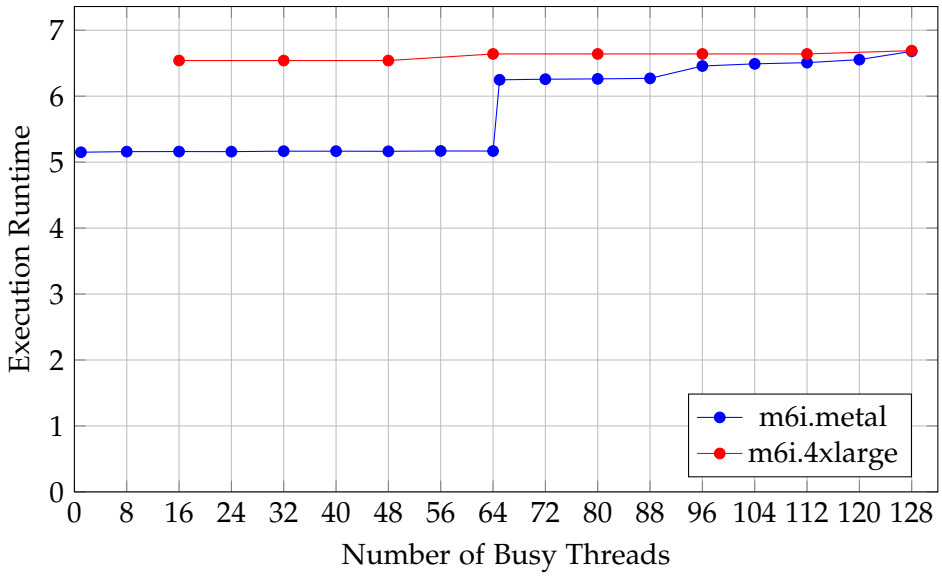


Figure 5.8: Performance of the test node (while incrementally adding busy neighbors) in comparison to running the threads natively on m6i.metal

At 128 threads, both the m6i metal and the test nodes had the same execution runtime, which further supports the claim that the difference between busy and idle neighbor in Table 5.4 is not solely due to hypervisor overhead. We notice the same behavior we witnessed on 1st gen and 2nd gen Intel Xeon Scalable processor from the m5.family. We notice that in both the m5.metal and m6i.metal, the most significant performance degradation happens exactly at $n/2+1$ with n being the maximum number of hyper-threads available. Our assumption is that the first $n/2$ threads are scheduled each on an independent physical core. However the $n/2+1$ thread needs to share a physical core with another thread, as explained previously. This results in the performance degradation of 20.9% we witness here and 25.22% using m5.metal. The maximum performance degradation using this m6i.metal is less than m5.metal, reaching 31.85% here in comparison to 53%. Over the first 64 threads ($n/2$), we notice very small performance degradation of 0.35%, in comparison to 13.2% on the m5.metal instance (over the first 48 threads). In this experiment as well, we notice that the the first m6i.4xlarge has an intial performance 26.7% worse than deploying the threads natively on the bare-metal instance. We assume that the same vCPU distribution happens with this host, as explained for the m5 host.

5.3.3 m6a Family

The m6a host features the AMD EPYC 7R13 Processor. Table 5.5 captures the maximum achievable performance degradation on the test node while adding busy or idle neighbors.

Instance type	large	xlarge	2xlarge	4xlarge
Maximum Nodes	96	48	24	12
Degradation (Busy) %	10	9.5	11.1	11.4
Degradation (Idle) %	0.05	0.06	0.06	0.05

Table 5.5: Maximum achievable performance degradation on our test node across various m6i instance types

When adding busy neighbors, we notice close percentages of performance degradation across the different instance types, with an average of 10%. When alone on the dedicated host, the test node started from roughly the same initial nominal value across all the instance types (7.09 seconds). Figure 5.9 compares the runtime on our test node when adding busy m6a.4xlarge neighbors to the runtime of running the threads natively on the m6a.metal instance.

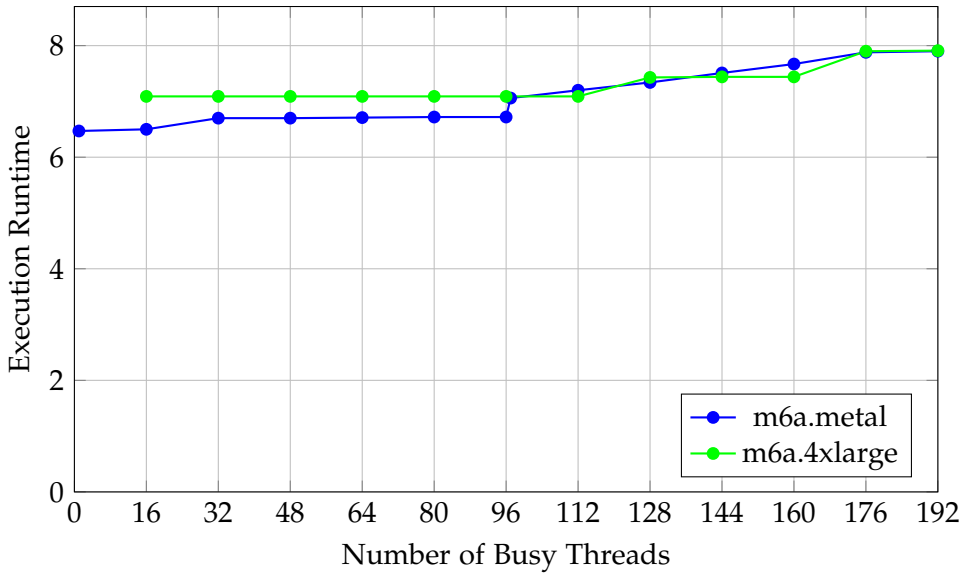


Figure 5.9: Performance of the test node (while incrementally adding busy neighbors) in comparison to running the threads natively on m6a.metal

The overall performance degradation for the metal instance is equal to 23.5%. For the first 96 threads, we observe a performance degradation of 3.9%. When adding the 97th thread ($n/2+1$), we witness an additional degradation of 5.2%, which is less significant than the degradation we saw at the same level for m5 and the m6i dedicated hosts (respectively 25% and 20%). We observe that the majority of the degradation occurs in the second half of adding the busy threads, i.e., from thread 97 to 192. During this phase, the degradation increases almost steadily from 9% to 23%. The degradation on the second half was also present on the m5 and m6i host with respectively 8.63% and 7%. The first m5.4xlarge (test node) had an initial runtime 9.3% worse than running *cpu_burn* with 16 threads on the m6a.metal instance, which implies the same mapping of the vCPUs across the physical cores as discussed previously.
We have now established that the witnessed degradation in the different families is not caused by physical core co-location, as the vCPUs of each VM are isolated. It is also unlikely that it is caused by virtualization overhead, as the performance of the metal instance and the VM converge toward the same value at full capacity. In the following part, we investigate the role of frequency scaling in these performance losses (H3).

5.4 Contention under Single Threaded Core Processors

5.4.1 m7a Family

The m7a host, powered by the AMD EPYC 9R14 processor, utilizes frequency scaling and has SMT disabled, making it an ideal candidate for investigating the effect of frequency scaling on degradation. We ran the CPU experiment using 4xlarge instances to determine whether any degradation occurs on the test node. As table 5.6 shows, we observe a performance loss of 9.15% by adding busy neighbors. The metal instance experiences a similar degradation of about 9.3%. For the metal instance, the initial runtime corresponds to running *cpu_burn* with 16 threads and final runtime with 192 threads.
Since virtual machines do not have read access to the frequencies of each individual cores, we measured the average CPU frequency of the busy cores on the metal instances. Busy cores can be identified by their frequencies exceeding 2.6 GHz, which corresponds to the base clock of the AMD EPYC 9R14 processor. With 16 busy threads, we measured an average frequency of 3.7 GHz, which represents the peak frequency. However, at 192 busy thread, we observe a drop to 3.4 GHz in the average frequency (8.1%).
These findings strongly suggest that the observed performance degradation is caused by the frequency scaling mechanism: As more physical cores become busy, the average frequency decreases to adhere with power consumption and temperature limits. This explanation also extends to the previous experiments, since all hosts supported frequency scaling.

	test node (4xlarge)	**m7a.metal**	**Average frequency**
Initial runtime (s)	6.642	6.66	3.7 GHz
Final runtime (s)	7.25	7.28	3.4 GHz
Degradation	9.15%	9.3%	8.1%

Table 5.6: Runtime and frequency comparison for the test node (4xlarge) and the m7a.metal instance

5.4.2 m6g Family

We now examine CPU contention on the m6g dedicated host that runs on the AWS Graviton2 processor. It features the Nitro 2 Hypervisor, the same as the m5 family. The host has 64 physical cores and therefore 64 vCPUs. Table 5.7 summarizes the instance types belonging to this family.

Instance Type	vCPUs	RAM (GiB)
m6g.medium	1	4
m6g.large	2	8
m6g.xlarge	4	16
m6g.2xlarge	8	32
m6g.4xlarge	16	64
m6g.8xlarge	32	128

Table 5.7: vCPU and RAM specifications for m6g instance types

We ran the CPU experiment across different instances types and summarized the results in Table 5.8.

Instance type	medium	large	xlarge	2xlarge	4xlarge
Maximum Nodes	64	32	16	8	4
Degradation (Busy) %	0.05	0.03	0	0	0

Table 5.8: Maximum achievable performance degradation on our test node across various m6g instance types

The results show minimal degradation, which is likely caused by virtualization overhead. As advertised by AWS, the Nitro system causes practically no overhead (sub 0.05%) and performance is almost indistinguishable from bare-metal. Figure 5.10 further supports this. It compares the runtime of the test node while adding busy m5.2xlarge neighbors to the runtime of running the threads natively on the m6g.metal instance. We can see that both graphs are practically similar. This behavior was expected, since Graviton processors provide isolation between the different threads, with each thread running independently on a physical core. Graviton processors also do not support frequency scaling, so no degradation is caused by a reduction in average frequency.

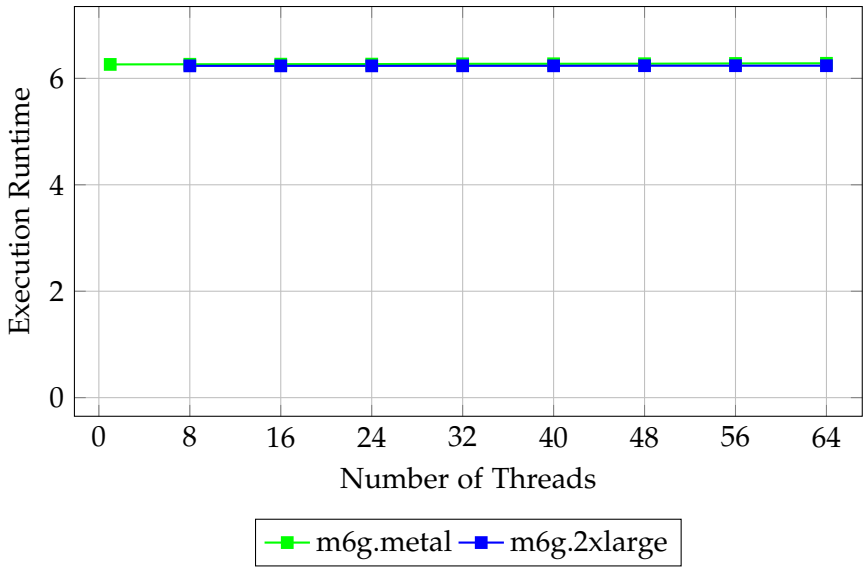


Figure 5.10: Effect of adding threads on the CPU performance using m6g.metal and the cpu_burn tool

5.5 Discussion

The m5 instances are prone to CPU contention, with performance degradation ranging from 3.25% (4xlarge) to 13% (large) (Table 5.3). For this family in particular, experiments show that degradation caused by adding idle neighbors is on par with the degradation from adding busy neighbors. Han et al. [12] observed this behavior and were able to improve sysbench performance by 20.81% on an m5d.large instance by shutting down idle VMs. This behavior seems unique to the m5 family as they could not replicate it on the c3 and c4 families.
The m6i family showed only minor degradation with values around 2%. In contrast, the m6a instances showed more important degradation with values reaching 11%. In both families, idle instances practically did not affect the test node's performance.
Han et al. [12] suggested that context switching overhead introduced by the Nitro scheduler could create performance degradation. However, this explanation is implausible, since the Nitro System provides a near bare-metal performance as seen for the m6g family (see Figure 5.10). Thus, our first hypothesis is accepted. The second hypothesis is rejected, since we found that instances with n vCPUs running on an SMT-enabled host only have access to $n/2$ physical cores, which excludes any possible degradation caused by physical core co-location between different co-tenants. This distribution, however, affects the initial performance of deployed VMs, as they cannot benefit from

non-occupied cores. This is reflected in the initial difference between the runtime of the first deployed instance on the dedicated host and the runtime of running the threads directly on the metal instance. This difference reached 36% on m5, 27% on m6i, and 9.3% on m6a using large instances.
We argue that the observed performance degradation when adding busy neighbors is caused by the frequency scaling feature. We confirmed this in our experiment with the m7a family, where we measured a decrease in the average frequency of the busy cores as the number of busy threads are increase (see Figure 5.6). This finding validates the third hypothesis. For the m6g dedicated host, we witnessed almost no CPU contention with degradation values lower than 0.05%. This behvaior was expected, since Graviton processors do not support frequency scaling, and the virtualization overhead is minimal.

6 Network I/O Contention

6.1 Throughput Contention

6.1.1 Methodology

Unlike other resources such as CPU and RAM, which are statically divided between the tenants based on instance type, network bandwidth is shared among the different co-tenants without a fixed specification of the expected bandwidth per tenant. Typically, for instances with 16 vCPUs or fewer, AWS specifies the bandwidth upper bound, e.g., "Up to 10 Gbps" [1]. However, these instances still have a baseline bandwidth. A network I/O credit mechanism allows the instance to use burst bandwidth for a short period, ranging from 5 to 60 minutes, depending on the instance type [1].
Bandwidth throttling for smaller instances takes at least 5 minutes to be enforced, during which the instance has access to 10 Gbps burst bandwidth. We conduct our experiments in this time window. It is particularly interesting to observe the extent of the network degradation in comparison to the baseline bandwidth for each instance size.
For network I/O stress, we used iPerf3 [16], a tool that benchmarks network bandwidth. It supports multiple protocols and can measure TCP, UDP, and SCTP throughput. iPerf3 probes the maximum achievable network bandwidth by transmitting a large number of packets until the thoughput's upper bound is reached. The tool requires two nodes — a server and a client. Our benchmark measured the maximum UDP throughput. We chose UDP in order to avoid congestion effects introduced by TCP congestion control, which can reduce measured throughput even when additional bandwidth is available. To quantify throughput degradation caused by contention between the different tenants, we conducted the following experiment.

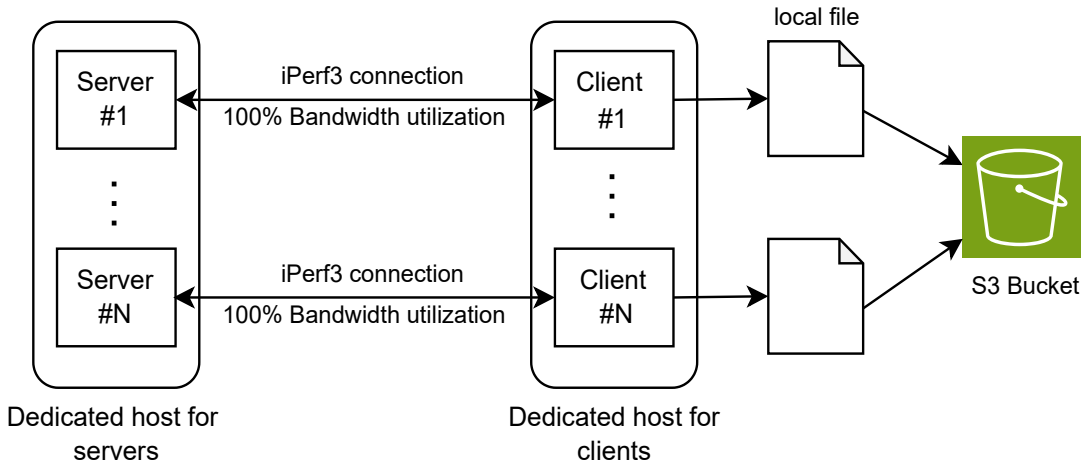


Figure 6.1: Throughput contention experiment

We deployed two dedicated hosts: one hosting the iPerf3 client instances and the other hosting the corresponding server instances. We incrementally added client nodes that fully utilized their bandwidth through iPerf3 connections with their paired servers. Each client continuously recorded its measured throughput to a local log file. We implemented a Python script that computes the average of the 20 recent data points in the log file and appends the result to a separate output file. After each client was deployed, the script was executed across all client nodes using the Distexprunner tool. At the end of the experiment, all the output files from the clients were uploaded to an S3 bucket for further analysis.
Single flow traffic is limited to 5 Gbps, for instances residing in different cluster groups. This is the case for the clients and servers since they are placed on different dedicated hosts. To bypass this limitation, we established two client connections using the *-P* option in the iPerf3 tool. Additionally, the command duration was set to 3600 seconds (using the `-t` option) to ensure steady-state network performance without fluctuations caused by repeatedly calling the command.
All the VMs were provisioned in the same availability zone and resided in the same Virtual Private Cloud as mentioned in the Testing Infrastructure chapter.

6.1.2 m5 Family

This subsection analyzes throughput contention on instances belonging to the m5 family. KPIs for the m5 dedicated host can be found in Table 5.1. Table 6.1 depicts the most important network-related specifications for the different instance types.

Type	vCPUs	Burst BW (Gbps) [26]	Baseline BW [26]
m5.large	2	10	0.75
m5.xlarge	4	10	1.25
m5.2xlarge	8	10	2.5
m5.4xlarge	16	10	5
m5.8xlarge	36	10	10
m5.12xlarge	72	12	12
m5.metal	96	25	25

Table 6.1: Specifications of m5 instance types (BW = Bandwidth)

The first experiment features the m5.4xlarge instance type. The maximum number of VMs on the host is 6. To provide a clearer visualization of the results, we present each node in a separate graph in Figure 6.2.

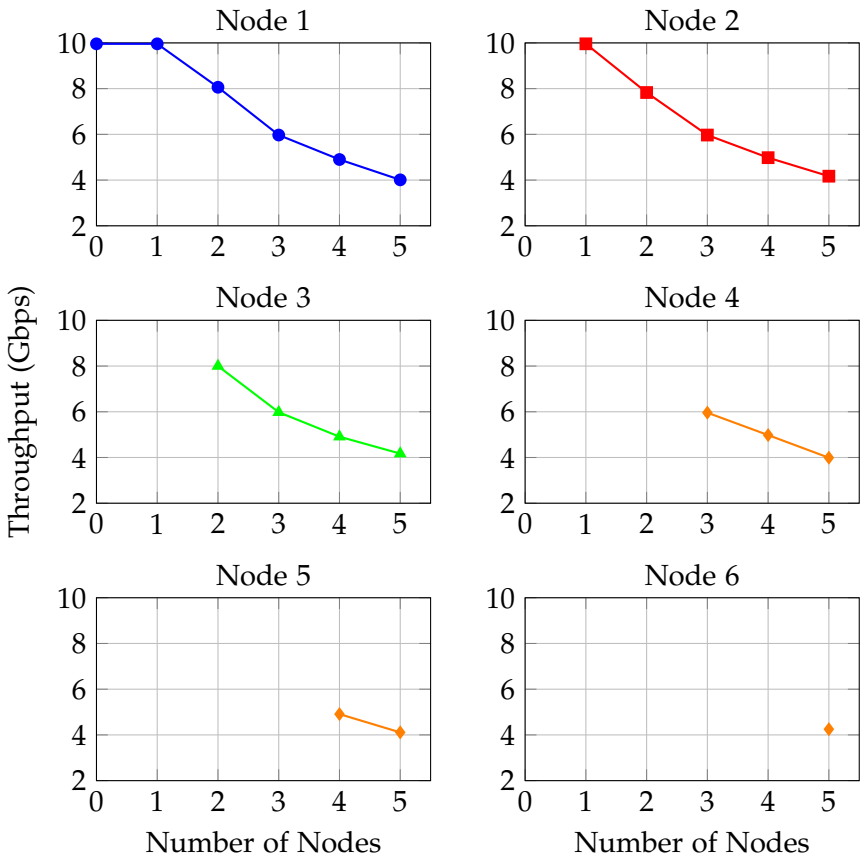


Figure 6.2: UDP Throughput of m5.4xlarge nodes when incrementally increasing the busy tenants

The first and the second clients had access to a burst bandwidth of 9.96 Gbps. The addition of the third tenant caused the average throughput to drop to 7.96 Gbps, while the fourth neighbor further reduced it to 5.96 Gbps. The fifth neighbor decreased the throughput of all co-tenants to roughly the baseline bandwidth of the m5.4xlarge (5 Gbps) with an average throughput of 4.94 Gbps. The sixth node introduced the first significant decrease below the baseline bandwidth to an average of 4.12 Gbps, which is 17.6% less than the baseline. Starting from the third tenant, the sum of the throughputs across all nodes remained around 24.7 Gbps. This was expected as the bandwidth of the m5.metal is 25 Gbps, representing the upper limit for the sum of the throughputs of all the nodes residing on the same dedicated host.
For the xlarge, 2xlarge, and 4xlarge types, the product of the maximum number of nodes on the dedicated host multiplied by the baseline bandwidth of the respective

instance type equals 30 Gbps. This value is 16.7% smaller than the 25 Gbps bandwidth of the physical server, which explains the average degradation of 17% we observed in the previous experiment. The same behavior should be expected when using xlarge and 2xlarge instances. For the large instance type, however, the product is equal to 36 Gbps (48×0.75). 25 Gbps is 30% smaller than 36 Gbps, indicating that a degradation of around 30% is anticipated at full capacity when using m5.large instances. We verify this assumption in the next experiment. Since the dedicated host can host 48 of m5.large instances, we cannot plot the graph of each instance. Instead, we used a plotbox graph to display the distribution of the throughputs at different occupancy levels, as shown in figure 6.3.

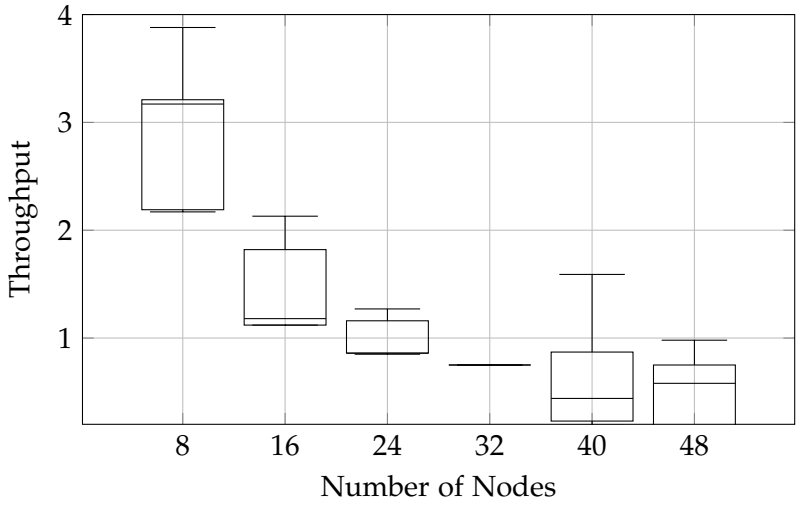


Figure 6.3: Throughput (UDP) of m5.large nodes when incrementally increasing the number of busy tenants

The first node has access to a burst bandwidth of 9.96 Gbps. At eight neighbors, the average throughput dropped to 3 Gbps. We then observe a gradual degradation with an average of 1.5 Gbps at 16 nodes, 1 Gbps at 24 nodes, and 0.75 Gbps at 32 nodes, which corresponds to the baseline bandwidth of the m5.large instance type. At this level, we observed virtually no variation between the throughput levels. At 40 nodes, the average bandwidth decreased to 0.614 Gbps and further to 0.51 Gbps at 48 nodes. At full capacity, The average throughput (0.51 Gbps) was 30.7% lower than the baseline bandwidth of the m5.large instance (0.75 Gbps). This more pronounced performance degradation close to 30%, aligned with our earlier hypothesis. In this experiment, we also observed a notable performance variation between the different nodes compared to the previous experiment.

6.2 Latency

6.2.1 Methodology

For latency benchmarking, we employed both *iPerf3* and *sockperf* [24]. Sockperf is a network benchmarking utility capable of measuring the latency of packets with sub-nanosecond resolution. It introduces minimal overhead by leveraging the Time Stamp Counter (TSC) registers, which count the number of CPU cycles for measuring latency [24]. Sockperf also requires a client-server setup: the client sends mutliple packets to the server , receives responses and records the Round-Trip Time (RTT) for each packet. The tool provides different options to visualize the results with varying granularity. A CSV file is generated, listing the send and response times for each individual packet, along with a report of the average latency and key metrics such as the 90th and 99th percentile.
The objective of the latency experiment was to evaluate the effect of increasing aggregate throughput utilization of neighbors on the latency of the test node. The experiment is structured as follows.

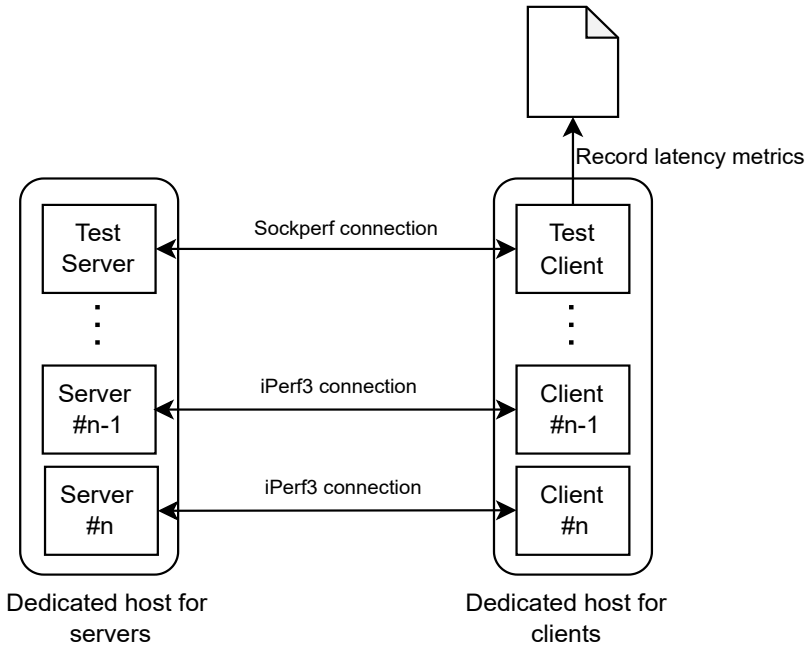


Figure 6.4: Latency degradation experiment

Similar to the throughput experiment, we deployed two dedicated hosts, one host-

ing all clients and one hosting all servers. Initially, only the client test node and its corresponding server were created. We measured the latency between them and recorded it as a baseline value. Subsequently, we deployed all remaining clients and incrementally increased their aggregate throughput in 1 Gbit/s steps (or smaller when latency degradation was observed). This was achieved through the iPerf3 `-b` option, that allows specifying an exact throughput value for the client, enabling precise control over the aggregated throughput of the neighbors. After each step, we measured the latency between the test client node and the test server using sockperf. The results were recorded locally on the client test node. Since all dedicated hosts, including the client test node and server, are located within the same Availability Zone (AZ) (See Chapter 4), external latency variability was minimized, allowing for a better identification of latency degradation caused by increased throughput usage of the neighbors.

6.2.2 m5 family

The first experiment was conducted on the m5 dedicated host. At each throughput level, we executed the sockperf command 30 times and averaged the metrics recorded. We extracted the average latency as well as the 90th and 99.9th percentile values. Figure 6.5 shows the results of the experiment.

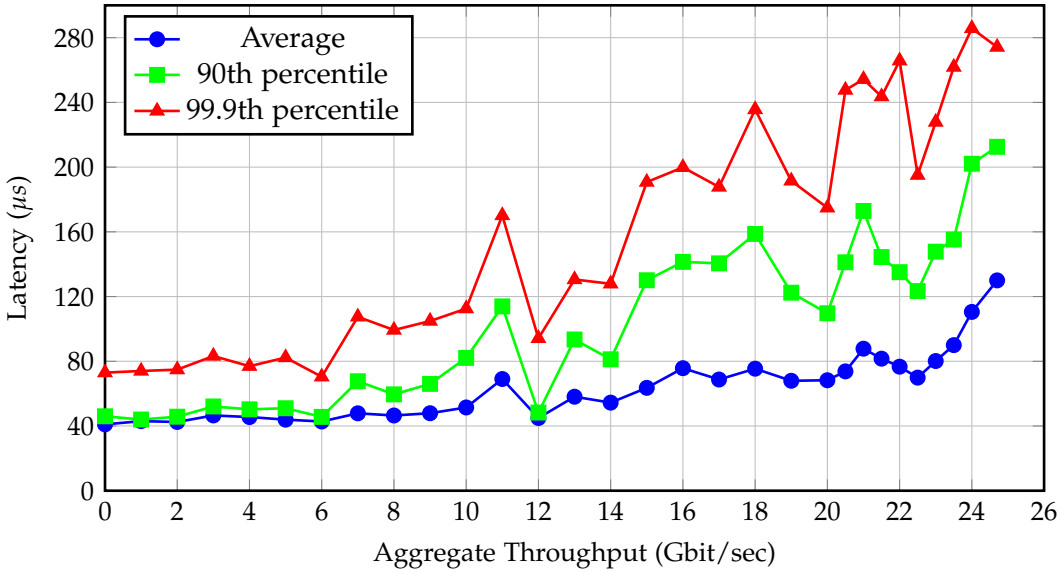


Figure 6.5: Average Latency vs Bandwidth

The baseline average latency between the test client and the server was 41 μs, which

is relatively low considering that both reside in the same AZ. From the start of the experiment up to an aggregate throughput of 9 Gbit/s, latency exhibited only minor degradation, increasing by 11.9% to 47 μs. Beyond this point, latency began to rise significantly in a gradual manner that became sharper, as the aggregate throughput approached the maximum aggregate throughput of 24.7 Gbit/s. It increased from 51 μs at 10 Gbit/s to 130 μs at 24.7 Gbit/s , corresponding to an 155% increase. Overall, the experiment revealed a total latency degradation of approximately 217%, a substantial value that could have severe implications for latency-sensitive workloads.
Furthermore, the results also indicate siginificant variability. The 90th percentile latency values were, on average, 41 μs higher than the mean, while the 99.9th-percentile values exceeded the mean by 100 μs, on average. This variability became more pronounced as the mean latency degraded. Figure 6.6 illustrates the difference between the mean and the 90th-percentile latency value, clearly highlighting the increased variability with increased aggregate throughput.

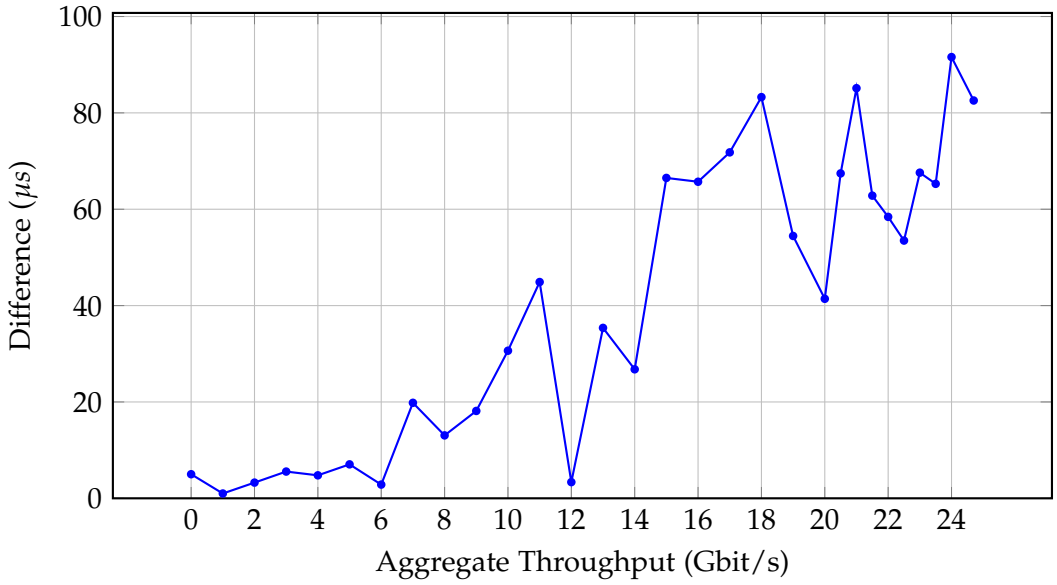


Figure 6.6: Difference between the 90th-percentile latency values and the mean

6.3 Discussion

For throughput, we found out that the (average) maximum degradation for any instance type can be approximated by the following formula. The bandwidth of the physical server is equal to the bandwidth of the metal instance of the concerned family (if it's offered):

$$\frac{\text{Bandwidth of physical server / Maximum number of nodes}}{\text{Baseline Bandwidth of the instance type}} \times 100$$

For all the instance types belonging to the m5 family except the large type, the maximum degradation relative to the baseline bandwidth is roughly 17%, whereas for the large type it reaches 30%. In their work, Loyd et. al [12], found similar results using the m5d dedicated host, which is similar to the m5 host, as they both have the same processor and number of vCPUs. For large instances, they observed a normalized performance degrdation equal to 94.6% due to VM co-location. However this percentage is relative to the inital throughput of the first m5d.large, i.e., the burst bandwidth of 10 Gbps. It implies that the average throughput at the end was 0.54 Gbit/s, very close to the results presented in Figure **??**. Lloyd et. al also found that degradation tends to increase with successive VM generation, which can be explained by denser and larger physical servers capable of hosting an increasing number of tenants.
For latency, We measured an average degradation of 217% with increasing aggregate throughput of the neighbors. We used the m5.4xlarge instance for our experiment. However, the results should be independent of the instance type, as the only influential factor is the aggregate throughput of the neighboring nodes. It's important to note that the datapoints exhibited very high variability and the results can significantly different between different runs. The percentile values, increasingly far from the mean, support this.

7 Conclusion

This thesis contributes empirical evidence of the worst-case impact of VM co-location on AWS instances, with a focus on CPU and network I/O. For CPU, we found that SMT-enabled hosts are particularly susceptible to performance degradation. Among the evaluated families, m5 and m6a were the most affected, with performance loss reaching 13%. In contrast, the m6i family exhibited only minor degradation of less than 2%. The m5 family also showed unexpected performance degradation even with idle neighbors. Furthermore, our results suggest that AWS enforces vCPU isolation between virtual machines, that has a negative impact on the baseline performance of these instances.

These findings highlight that even with continued improvement to virtualization, considerable resource degradation is still possible. Hardware features such as SMT can introduce non-trivial performance penalties. It's crucial that AWS customers consider these characteristics when choosing an instance type for a CPU-intensive workload. m5 instance users seem to be at a disadvantage, since CPU performance is degraded solely by the presence of neighbors. However, Graviton instances, such as the m6g family, provide a (nearly) complete isolation between tenants, offering consistent and reliable CPU performance.

The thesis also investigated network I/O contention on the m5 family and its effect on throughput and latency. Throughput degradation can reach 30% for large instances, relative to their baseline bandwidth. For latency, the measurements had high variability but we discovered that a degradation of 217% is possible.

Throughput and latency degradation are mostly dependent on the bandwidth of the underlying dedicated host. For network-intensive applications, users should opt for network-optimized instances that provide higher bandwidth. Otherwise, co-location effects could severely impact application performance.

Abbreviations

SMT Simultaneous Multi-Threading

AWS Amazon Web Services

VMM Virtual Machine Monitor

IaaS Infrastructure-as-a-Service

IaC Infrastructure-as-Code

HCL HashiCorp Configuration Language

RTT Round-Trip Time

AZ Availability Zone

Bibliography

[1] Amazon Web Services. *Amazon EC2 Instance Network Bandwidth*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html. Accessed: 2025-07-14. 2025.

[2] Amazon Web Services. *Amazon EC2 Instance Types - General Purpose Instances*. https://docs.aws.amazon.com/ec2/latest/instancetypes/gp.html. Accessed: 2025-06-20. 2025.

[3] Amazon Web Services. *Amazon EC2 M5 Instances*. Accessed: 2025-06-12.

[4] AMD. *AMD Ryzen™ Technology: Precision Boost 2 Performance Enhancement*. https://www.amd.com/en/resources/support-articles/faqs/CPU-PB2.html. Accessed: 2025-08-19.

[5] *AWS EC2 Instances – Vantage Instances*. https://instances.vantage.sh/. Accessed: 2025-08-02.

[6] J. D. Bean et al. *The Security Design of the AWS Nitro System*. Tech. rep. Amazon Web Services (AWS), Nov. 2022.

[7] P. S. Foundation. *csv — CSV File Reading and Writing*. https://docs.python.org/3/library/csv.html. Python 3.13.5 documentation; accessed 2025-08-02.

[8] P. S. Foundation. *re — Regular expression operations*. https://docs.python.org/3/library/re.html. Python 3.13.5 documentation; accessed 2025-08-02.

[9] Gentoo Wiki contributors. *Sysbench*. https://wiki.gentoo.org/wiki/Sysbench. Accessed: 2025-05-25.

[10] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. "Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines." In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. SOCC '11. Cascais, Portugal: Association for Computing Machinery, 2011. ISBN: 9781450309769. DOI: 10.1145/2038916.2038938.

[11] B. Gregg. *AWS EC2 Virtualization 2017: Introducing Nitro*. Brendan Gregg's Blog. Nov. 2017.

[12] X. Han, R. Schooley, D. Mackenzie, O. David, and W. J. Lloyd. "Characterizing Public Cloud Resource Contention to Support Virtual Machine Co-residency Prediction." In: *2020 IEEE International Conference on Cloud Engineering (IC2E)*. 2020, pp. 162–172. DOI: 10.1109/IC2E48712.2020.00024.

[13] K. Hess. "Understanding Hardware-Assisted Virtualization." In: *ADMIN Magazine* (Aug. 2011).

[14] M. S. Inci, B. Gulmezoglu, T. Eisenbarth, and B. Sunar. "Co-location Detection on the Cloud." In: vol. 9689. Apr. 2016, pp. 19–34. ISBN: 978-3-319-43282-3. DOI: 10.1007/978-3-319-43283-0_2.

[15] Intel Corporation. *What Is Intel Turbo Boost Technology?* https://www.intel.com/content/www/us/en/gaming/resources/turbo-boost.html. Accessed: 2025-08-19.

[16] iPerf Project. *iPerf – The TCP, UDP and SCTP Network Bandwidth Measurement Tool*. https://iperf.fr/. Accessed: 2025-06-01.

[17] A. Kopytov. *sysbench: Scriptable database and system performance benchmark*. https://github.com/akopytov/sysbench. Accessed: 2025-05-23.

[18] W. Lloyd, S. Pallickara, O. David, M. Arabi, and K. Rojas. "Mitigating Resource Contention and Heterogeneity in Public Clouds for Scientific Modeling Services." In: *2017 IEEE International Conference on Cloud Engineering (IC2E)*. 2017, pp. 159–166. DOI: 10.1109/IC2E.2017.29.

[19] T. Moseley, J. Kihm, D. Connors, and D. Grunwald. "Methods for modeling resource contention on simultaneous multithreading processors." In: *2005 International Conference on Computer Design*. 2005, pp. 373–380. DOI: 10.1109/ICCD.2005.74.

[20] A. Rashid and A. Chaturvedi. "Virtualization and its Role in Cloud Computing Environment." In: *INTERNATIONAL JOURNAL OF COMPUTER SCIENCES AND ENGINEERING* Vol.-7 (Apr. 2019), pp. 1131–1136. DOI: 10.26438/ijcse/v7i4.11311136.

[21] M. S. Rehman and M. F. Sakr. "Initial Findings for Provisioning Variation in Cloud Computing." In: *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. 2010, pp. 473–479. DOI: 10.1109/CloudCom.2010.47.

[22] A. W. Services. *Amazon EC2 Dedicated Hosts Pricing*. Accessed: 2025-08-02.

[23] A. W. Services. *AWS Graviton Performance Testing: Tips for Independent Software Vendors*. https://docs.aws.amazon.com/pdfs/whitepapers/latest/aws-graviton-performance-testing/aws-graviton-performance-testing.pdf. Accessed: 2025-07-19. 2021.

[24] M. Technologies. *sockperf: Network benchmarking utility over socket API*. `https://github.com/Mellanox/sockperf`. Accessed: 2025-08-02. 2025.

[25] D. Tullsen, S. Eggers, and H. Levy. "Simultaneous multithreading: Maximizing on-chip parallelism." In: *Proceedings 22nd Annual International Symposium on Computer Architecture*. 1995, pp. 392–403.

[26] TUM DIS. *CloudSpecs*. `https://tum-dis.github.io/cloudspecs/`. Accessed: 2025-08-02.

[27] A. Upadhyay. *Two Threads, One Core: How Simultaneous Multithreading Works Under the Hood*. Accessed: 2025-07-19. 2024. URL: `https://blog.codingconfessions.com/p/simultaneous-multithreading`.

[28] E. de Wit. *erdewit/distex: Distributed process pool for Python*. `https://github.com/erdewit/distex`. GitHub repository, accessed 2025-06-01. 2024.