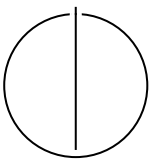
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

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

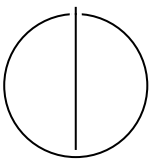
TECHNISCHE UNIVERSITÄT MÜNCHEN

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. Leis Viktor
Supervisor:	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

1 Introduction

Thrhtroughout the last years, infrastrcutre-as-a-service has become one of the most used business models. In this model, cloud providers abstract the server's physcial machiens and provide virtual machines to end-users that share these resources. This was possible thank to the drastic improvements that were introduced to the virtualization techonology with the intrdouction of hardware-assisted virtualization that significantly reduced virtualization overhead and allowed direct communicated of the guests with the virtualized hardware interfaces of the hardware. This, however, does not mitigated, perofrmance degradation that happens due to resource contention between the different tenants. In 2015, AWS introduced dedicated host, that enables cutomers to deploy VMs on a physical machines that's completely under their control. In this paper, we're interested on quantifying the extent of the perfromance degradation that can heppen and analyze its manifesation across key resources namely CPU and network.

2 Related Work

3 Background

3.1 Simultaneous Multi-threading

Before we dive deeper into simultaneous multithreading, it's important to understand which problem it actually 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 resource as busy as possible. To reach this, multiple techniques have been employed such as instruction pipelining, superscalar architecture and out-of-order execution, that improve instruction throughput and resource utilization of the processor. Pipelining is a technique that breaks down the execution of an instruction into several distinct stages, with each stage using separate hardware resources. 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 5th clock cycle, the CPU is simultaneously working on 5 instructions.

Clock Cycle / Instr. No.	**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 3.1: 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 simultaneously by dispatching them to different execution units. For example it's able to fetch two instructions on the same time.

Issue width is an important characteristic of modern CPUs and it represents the maximum number of instructions that can be executed simultaneously in a single clock cycle. Although these optimizations significantly increase the processor throughput, a relatively big independency between the instructions is required to be able to fully utilize the parallelism. Out-Of-Order execution partially solves this problem but is still not enough as it still dispatches instructions from the same thread, where the dependency between the instructions is generally high. The wastages that occur on the processor can be categorized into two categories: Horizontal waste and vertical waste. 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 because of their dependency to the currently executing instructions.

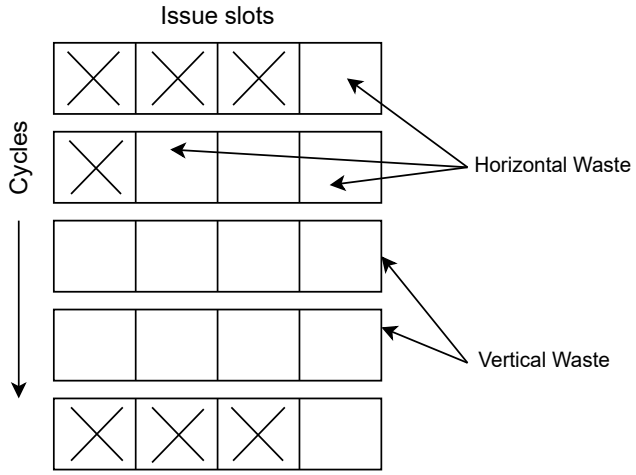


Figure 3.2: Vertical waste vs. horizontal waste

This is where Simultaneous Multi-Threading (SMT) comes into play. SMT is a technique that helps enhance the overall efficiency of superscalar CPUs by improving the parallelization of computation [9]. This technology allows the physical core to dispatch instructions from more than one thread [5] without requiring a context switch, effectively transforming each physical core into two (or more) "logical" cores. The idea is that instructions from different threads provide greate independency, which results in a better utilization of the core's execution resources. To be able to achieve this, some resources of the processor are duplicated, e.g., those that store the architectural state such as registers and program counters. However, the logical cores still share the same

execution resources, which can create conflicts, especially if both threads have the same workload nature, e.g., both are float heavy [9]. This technology can improve CPU throughput by taking advantage of idle time that the core formerly spent waiting for other instructions to be completed because of the dependeny between them [5] e.g., when the core is waiting for data from memory after a cache miss.

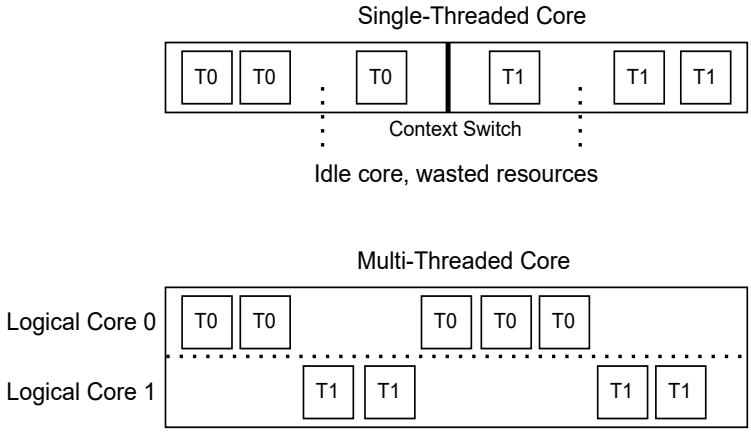


Figure 3.3: Single-Threaded Core vs. Multi-Threaded Core

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. The option to turn this feature off is always available via the BIOS/UEFI settings. In the AWS dedicated hosts that run on an Intel or AMD CPU with hyperthreading enabled, the number of vCPUs is always the double of the number of physical cores. 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, that are built around the ARM architecture, do not support hyper-threading [8] and expose one execution context, i.e., vCPU for each physical core. This allows for a better CPU isolation between the different tenants.

3.2 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. Each VM has its own operating system and acts as an independent physical computer. These VMs are called "guests" and the physical server is called "host". This technology is crucial

for the Infrastructure-as-a-Service (IaaS) model that's offered by cloud providers, as it allows for a greater resource- and cost-efficiency by dividing the physical server into different instance types, driving the price of compute resources down. Users can accordingly choose the instance type with the allocated resources that are optimal for their workload.
The main components that handle the necessary tasks for virtualization are the Virtual Machine Monitor (VMM) also called hypervisor and the management domain. Most of the instructions that are executed by Virtual Machines run natively on the CPU and do not require intervention from the VMM. However, when a privileged instruction, which is not available to regular applications, is encountered, the CPU raises a trap. The trap signals to the VMM to intervene and emulate the behavior of the instruction. After the emulation is finished, the control is then given back to the guest OS.
Virtualization cannot be accomplished 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 interface, storage drives, and input peripherals. Device models are required for this. They are basically software components that communicate with the shared hardware and expose multiple virtual device interfaces to the VMs. These 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. Since the device models are software-based, they compete for resources for CPU and system resources along with the existing VMs and can negatively affect the performance of these guests. The following figure summarizes the architecture of a traditional virtualization systems.

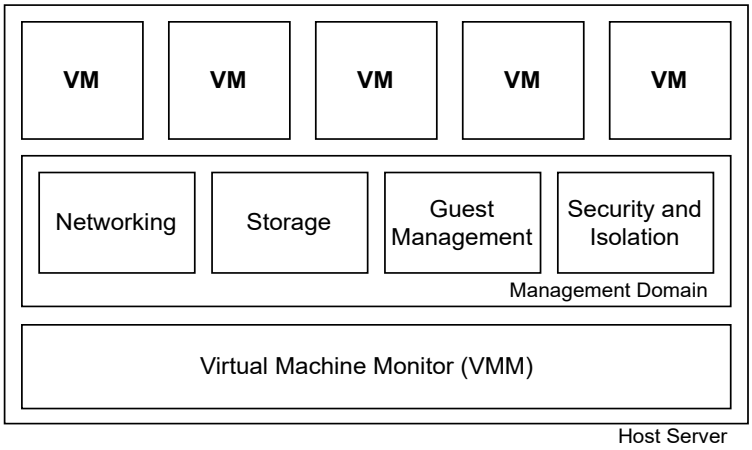


Figure 3.4: Architecture of traditional virtualization Solution

3.2.1 Evolution of Virtualization Solutions

Virtualization technology has evolved significantly. It began 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. This is, however, very slow and can make the host apps run 2x to 10x slower. Then paravirtualization was introduced, where the guest OS is modified to interact directly with the hypervisor via "hypercalls", removing the abstract emulation layer that is found in full software virtualization. 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. 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), which is a technology that allows physical PCIs device 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 thiese virtual devices, which increase I/O performance significantly.

3.2.2 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. The main idea was to decompose the software components that are 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. The following figure depicts the new AWS microservice architecture for virtualization.

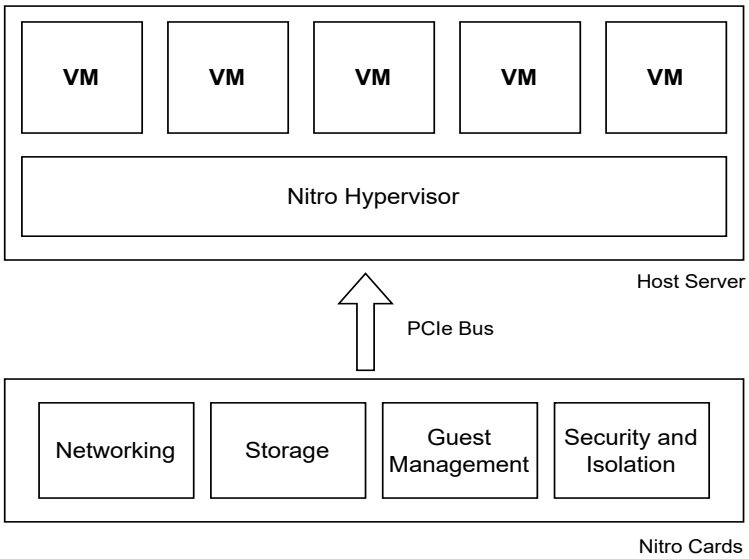


Figure 3.5: Architecture of Nitro System Virtualization

There are three main components in the AWS Nitro System.

The Nitro Cards

These 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 are responsible for executing all the tasks that concern the outside world in relation to the EC2 Server. They provide all I/O Interfaces such as the ones for storage and networking. When the host's system is using Nitro hypervisor, i.e., not a bare-metal, then these card employ the previously explained SR-IOV technology to provide direct hardware interfaces to the VMs. Example of nitrocards are Nitro card for EBS, Nitro Card for Instance storage, 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 main board including CPU and memory. It's managed by the Nitro Controller mentioned previously and plays a crucial role in enabling AWS to offer bare metal instances. 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.

The Nitro Hypervisor

The third component is the AWS Nitro Hypervisor. This hypervisor has much less responsibilities than normal hypervisors, as a lot of functions are offloaded by dedicated hardware. It has three main functionalities: partitioning memory and CPU by usiing the virtualization commands provided by the underlying processor, assigning the virtual hardware interfaces provided by the Nitro cards to the Virtual Machines, and handle the machine management commands that come from the Nitro Controller (start, terminate, stop etc.).

4 Methodology

To deploy the resources for the different experiments, we used terraform which is an Infrastructure-as-Code (IaC) tool that's developed by HashiCorp and can be used to define and provision resources using the HashiCorp Configuration Language (HCL). All the resources were deployed in the us-east-2 region. Additionally, we used distexprunner [10], which is a powerful tool written in python that helps write and run commands remotely across multiple nodes addressing them through their public IPs. Distexprunner was consistently used across all the benchmarks for various purposes but mainly to gather all the benchmark results from the different EC2 Instances into a central S3 bucket.

4.1 CPU Benchmark

To generate CPU stress, we used sysbench [7], which is a powerful cross-architecture tool, that can be used for CPU stressing, among other options. It performs the deterministic task of checking all prime numbers until reaching 10000 (default value) by doing standard division of the current number by all numbers between 2 and its square root [3]. The number of the worker threads can be specified as an argument. The tool allows the specification of the total number of events that should be performed by the created threads. We then use the total execution runtime as a comparison metric between the different experiments. For comparison purposes, we also developed our own CPU stressing tool called *cpu_burn* written in the 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 threads that will be created. It then returns the total wallclock runtime that was needed for the execution of this workload. The workload is defined in the following function. We compiled the program with the optimization level 0.

```
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 4.1: perform_work function in C

4.2 Network Benchmark

4.2.1 Throughput

For network I/O stress, we used iPerf [6]. This tool provides a benchmark for measuring the available network bandwidth. It supports various protocols and can be used to test TCP, UDP, and SCTP throughput. The tool probes the maximum achievable network bandwidth by transmitting a large number of packets until the upper limit of throughput is reached. In our experiments, we measured the maximum UDP bandwidth. We made this choice in order to avoid congestion effects that can be caused by TCP congestion control, which could reduce the throughput even though there still might be bandwidth available.

4.2.2 Latency

For latency benchmarking, we used the sockperf tool, which is a network benchmarking utility that can measure the latency of packets at a sub-nanosecond resolution. This tool introduces very low overhead as it uses Time Stamp Counter (TSC) registers that count the number of CPU cycles for measuring latency. iPerf and sockperf require two nodes to run, a server and a client. In our experiments, clients and servers were consistently deployed within the same Availability Zone, and private IP addresses were used.

5 CPU Resource Contention

5.1 m5 family

We start by analyzing CPU contention between nodes that run on dedicated hosts that support SMT. 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 [2]. The following table provides an overview of the different instance types that belong 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 5.1: m5 Instance Specifications [2]

The m5 dedicated host has 48 physical cores and therefore 96 vCPUs. It features the Nitro v2 Hypervisor [1]. We used terraform to deploy the resources in the us-east-2a zone. The experiment is structured as follows: We begin by deploying a node, referred to as test node on the dedicated host. Next, we incrementally add neighbors that fully utilize their CPUs. We analyze the effect of adding these neighbors on the runtime of running sysbench and cpu_burn on the test node. For our first experiment, we exclusively used m5.2xlarge instances, each featuring 8vCPUs and 32 GiB RAM. This means that the maximum number of nodes on the dedicated host is 12. In our experiments the dedicated host used the Cascade Lake-SP cpu (2nd gen). The results can be seen in the following figure.

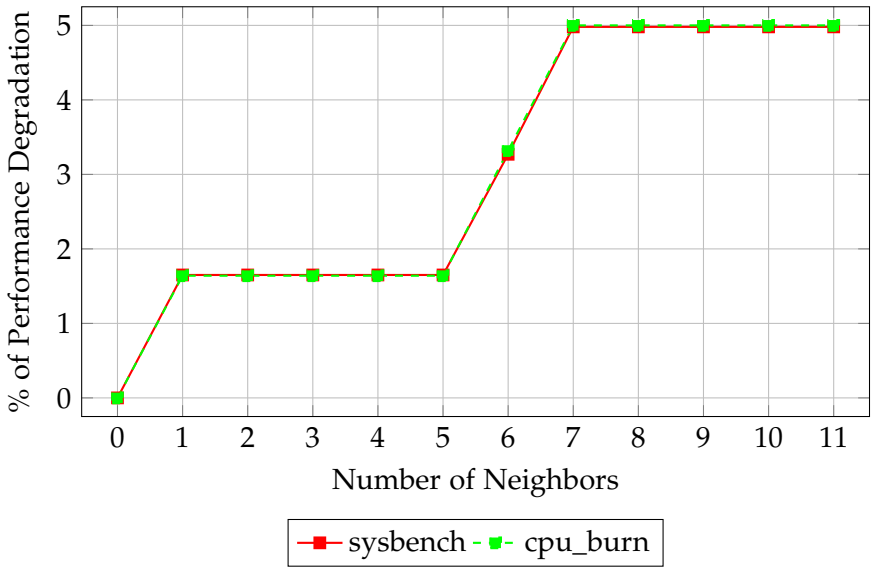


Figure 5.1: Effect of adding busy neighbors on the CPU speed of the sysbench and cpu_burn command on the test node using m5.2xlarge hosts

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 performance then remained constant form the next 4 neighbors. Afterwards, the 6th neighbor increased this degradation to 3.3%. The 7th 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, as it could be due to physical cores co-location between the different VMs or due to hypervisor overhead. We repeat the same experiment but we add idle VMs to see the extent of the performance degradation that happens. The results can be seen in the following figure.

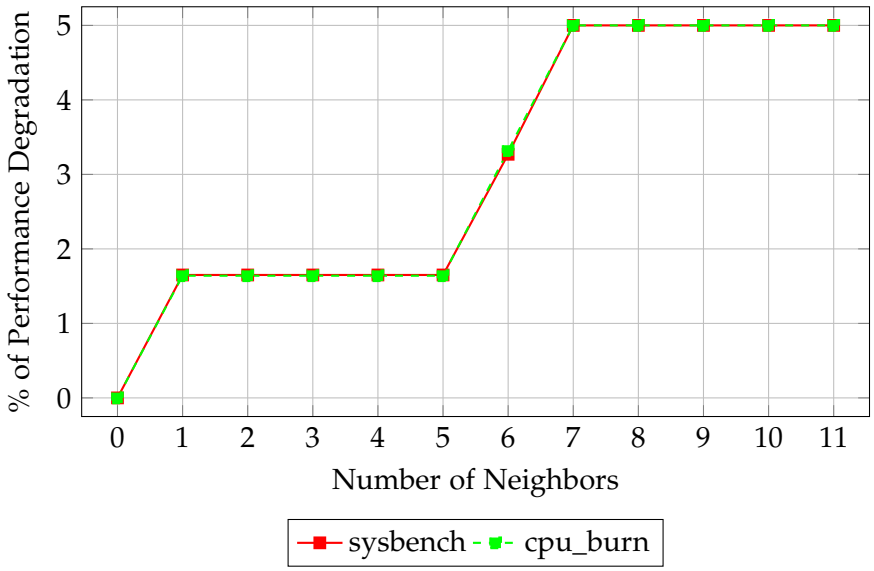


Figure 5.2: Effect of adding idle neighbors on the CPU speed of the sysbench and cpu_burn command on the test node using m5.2xlarge hosts (V)

We notice the exact same degradation pattern of the earlier experiment. This result strongly 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 investigate this problem 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 the following figure.

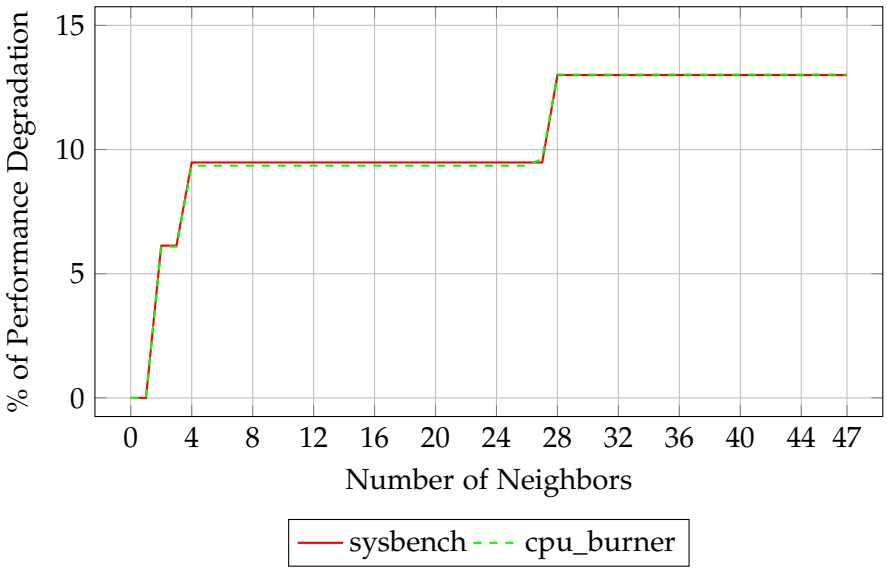


Figure 5.3: Effect of adding busy/idle neighbors on the CPU speed of the sysbench and cpu_burn command on the test node using m5.large hosts

In this experiment as well, we notice the same performance degradation 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. An important observation is that we notice nearly identical levels of performance degradation when using m5 hosts that run on 1st gen and 2nd gen Intel CPUs. Additionally, the results are always similar between the two tools. From this point, we'll only prcoceed with the cpu_burn tool. Adding idle or busy neighbors provides the same results in all our experiments. The results are summarized in the following table.

Instance type	large	xlarge	2xlarge	4xlarge	12xlarge
Maximum Nodes	48	24	12	6	2
Degradation %	13	13	4.8	3.25	0

Table 5.2: Maximum achievable performance degradation on our test node across various m5 instance types

The higher 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 5.1. 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, of which the dedicated host can only provision 2. In their paper, Han et. al. [4] argue that this CPU performance degradation is due to CPU context switching overhead that's caused by the KVM (Nitro) scheduler. Even though this hypothesis seems convincing, since the degradation decreases as the number of tenants decreases (Table 2), We think that it's very unlikely, as the Nitro Hypervisor should provide a near bare-metal performance. To investigate the problem 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 investigate whether we witness any performance degradation. The results can be seen in the following figure.

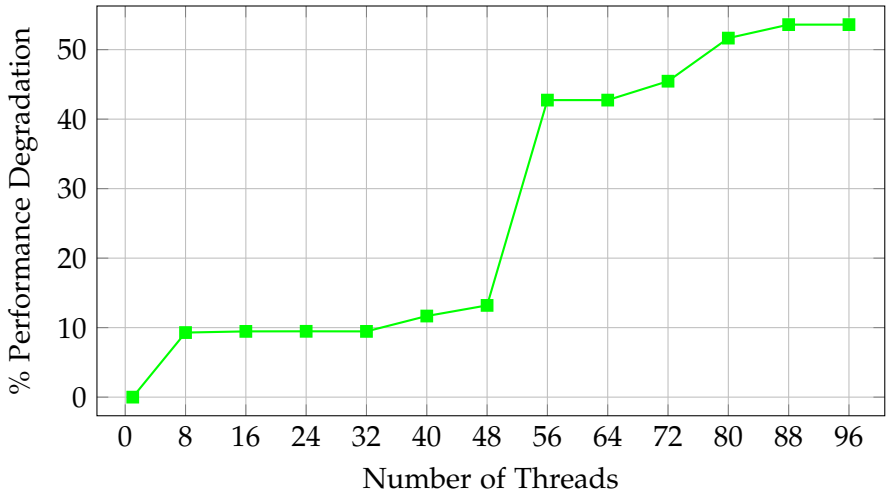


Figure 5.4: Effect of adding threads on the CPU performance using m5.metal and the cpu_burn tool

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 caused by the physical core co-location that happens between the different threads, resulting in resource contention, as the execution resources are not duplicated. We also notice a very interesting point that's worth mentioning. In all our experiments, the instances initially started with a baseline performance significantly worse than running the exact number of threads directly on the physical host. The following figure compares the runtime of cpu_burner on the test node (all threads are busy) as we

keep adding completely busy neighbors, in comparison to running the cpu_burner tool directly on the m5.metal while incrementally increasing the number of busy threads.

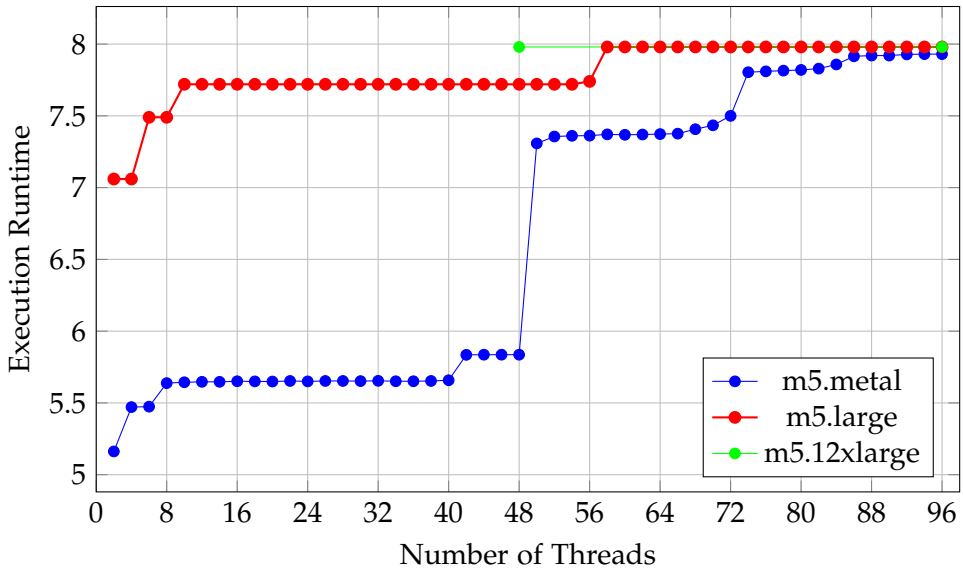


Figure 5.5: Performance of the test node in comparison to running the threads natively on m5.metal

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 (no other instances are deployed). 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 behind this is that the first 12xlarge instance started with a baseline performance that's 13% worse than the performance of the first m5.large instance. The bad 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 idle physical cores. We assume that this allocation

technique aims to avoid contention between the different tenants and isolate the vCPUs of each virtual machine by allocating each pair to the same physical core. To confirm our supposition, we run the cpu_burner tool in a m5.2xlarge instance, and incrementally increase the number of busy threads. The results can be seen in the following figure.

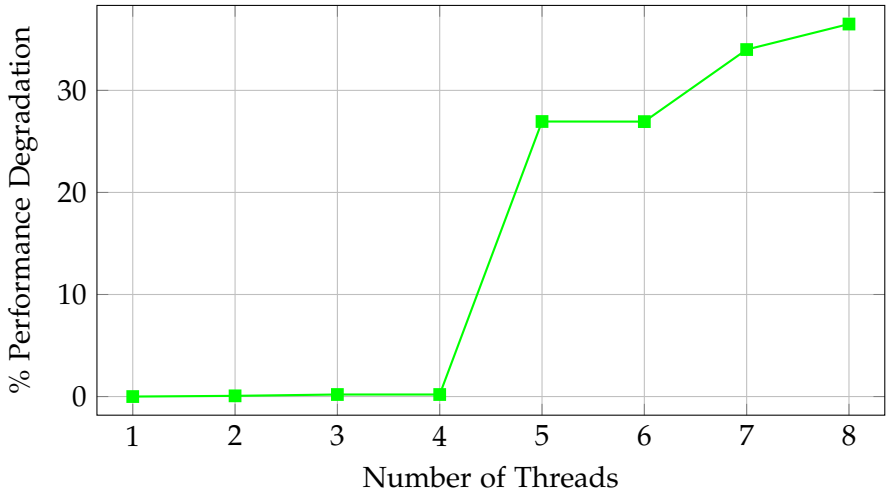


Figure 5.6: Effect of adding busy threads on the CPU performance using m5.2xlarge and the cpu_burn tool

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%.

Performance variation of random m5.large instances

We wanted to analyze the variation of the performance of different m5.large instances. For this we sequentially launched 50 m5.large instances across different zones of the us-east-2 region to see where their execution runtime is situated in relation to figure 5.5. The runtime across all the subjects was consistently 7.98 seconds. This consistency strongly suggests that AWS pre-provisions idle instances on internally managed dedicated hosts even before they're rented, enabling faster boot times when a customer actually rents a VM. If this were not the case, then we would have expected

to see execution runtimes around 7, 7.5 or 7.7 seconds which correspond to the three performance levels we witnessed on figure 5.5.

5.1.1 m6i family

It's interesting to see whether and to what extent the behavior we saw in the m5 family is present on the m6i family. The m6i family runs on the 3rd Generation Intel Xeon Scalable processor. We start directly by analyzing the execution runtime of parallel threads on the bare-metal offering of the m6i family, i.e., m6i.metal. The results of the experiment can be seen in the following figure.

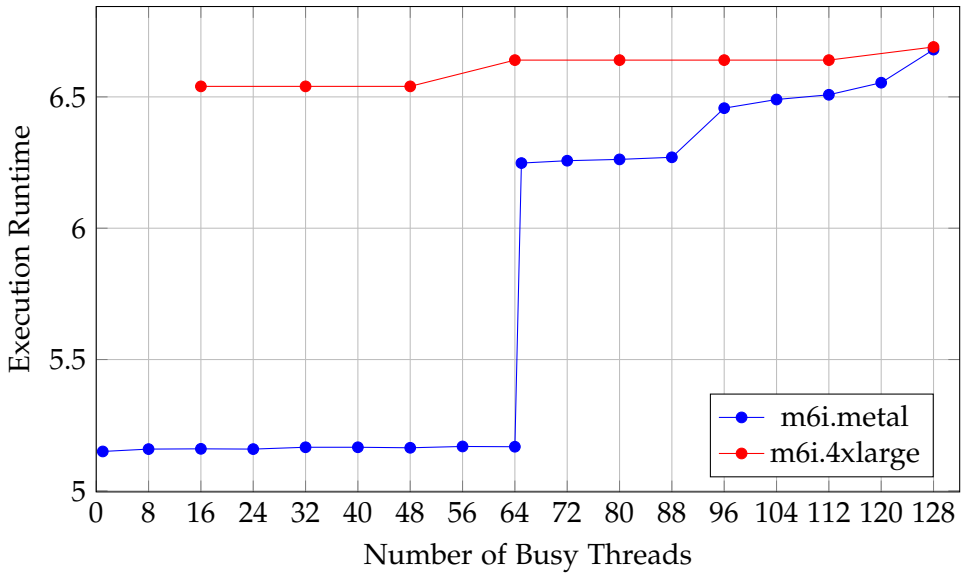


Figure 5.7: Performance of the test node in comparison to running the threads natively on m6i.metal

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 vCPUs available to the dedicated host. Our hypothesis 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 worse 26.7 % than deploying the threads natively on the bare-metal instance. We also investigated the maximum performance degradation that can happen on the different VMs and summarized the results in the following table.

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

Table 5.3: Maximum achievable performance degradation on our test node across various M5 instance types (V)

The difference here from the 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. these series of experiments can be misleading in suggesting that the performance degradation for the m6i family is better than m5 seeing the percentages. However, in these experiments, all the instance types started from nearly the same nominal performance level, which is only 2% away from the highest 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 in comparison to the other types.
It's very interesting whether this behavior also exists on other CPU that support multi-threading such as AMD. For this we repeat the experiment using the m6a instance.

5.1.2 m6a family

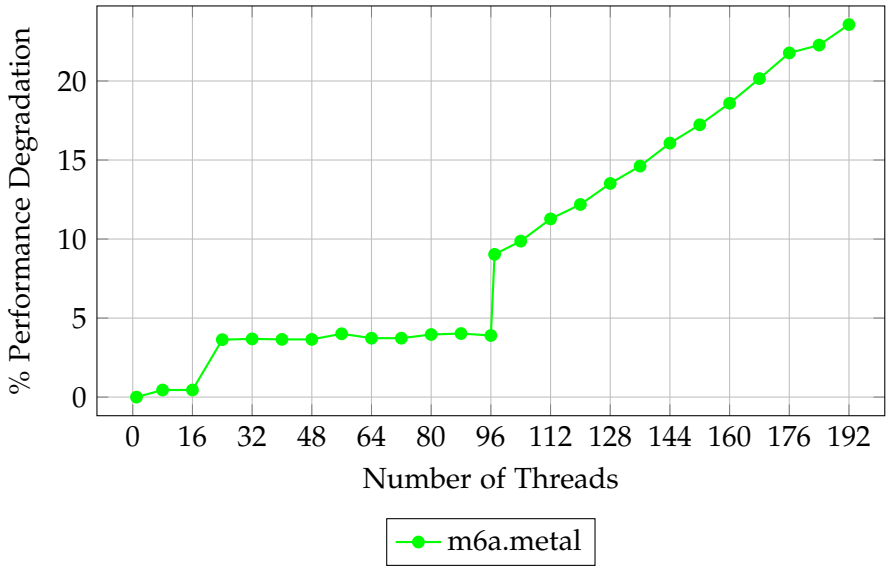


Figure 5.8: Effect of adding threads on the CPU performance using m6a.metal and the cpu_burn tool (V)

We notice a pattern similiar to that of the intel processors. However the maximum performance is less important and is equal to 23.5 %. When adding the 97th busy thread, we witness a degradation of 5.2%. We notice that the biggest part of the degradation happens in the second half of adding the busy threads, i.e., from thread 97 to 192. We see almost a steady upwards increasing line.
It's interesting to see the behavior of the virtual machines on the m6a dedicated host.

5.1.3 m6g family

We now examine CPU contention on hosts that do not support Hyper-Threading. For this, we used the m6g dedicated host that runs on the AWS Graviton2 pro- cessor. It also uses AWS Nitro 2 Hypervisor, which is the same as the m5 family. This host has 64 physical cores and therefore 64 vCPUs. The following table summarizes the instance types we used in the following experiments.

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 5.4: vCPU and RAM specifications for AWS m6g instance types

All the following experiments were conducted in the us-east-2a zone. We deployed a test node in the dedicated host and then incrementally added neighbors that are fully utilizing their CPU. We then analyze the effect of adding these neighbor on the CPU speed metric of the test node by comparing the runtime of the cpu_burn command. For our first experiment we used m6g.2xlarge nodes, of which the dedicated host can provision 8 instances. The results can be seen in the following figure.

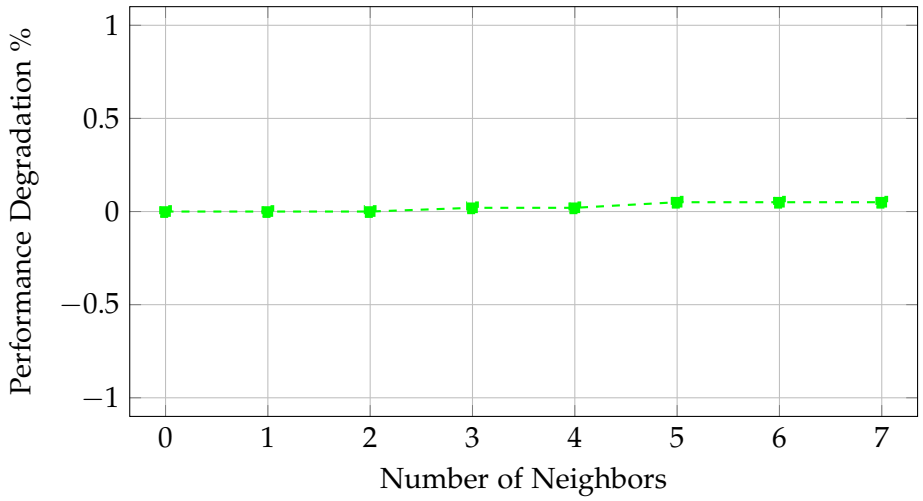


Figure 5.9: Effect of adding neighbors on the CPU performance with m6g.2xlarge instances using the cpu_burn tool (V)

We notice a very small and insignificant performance degradation of 0.05%. This should be due to hypervisor overhead which, as claimed by AWS, is practically non-existent. The following table captures the final performance degradation for different instances types. At each level, we repeated the cpu_burn command 10 times and then considered the average of these 10 values.

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

Table 5.5: Maximum achievable performance degradation on our test node across various M5 instance types (V)

The results of our experiment prove that the AWS Nitro hypervisor causes practically no overhead and the performance is almost indistinguishable from metal as advertised by AWS. We analyze the runtime of the different instance type in comparison to running the threads natively on the m6g.metal instance. The results can be seen in the following figure.

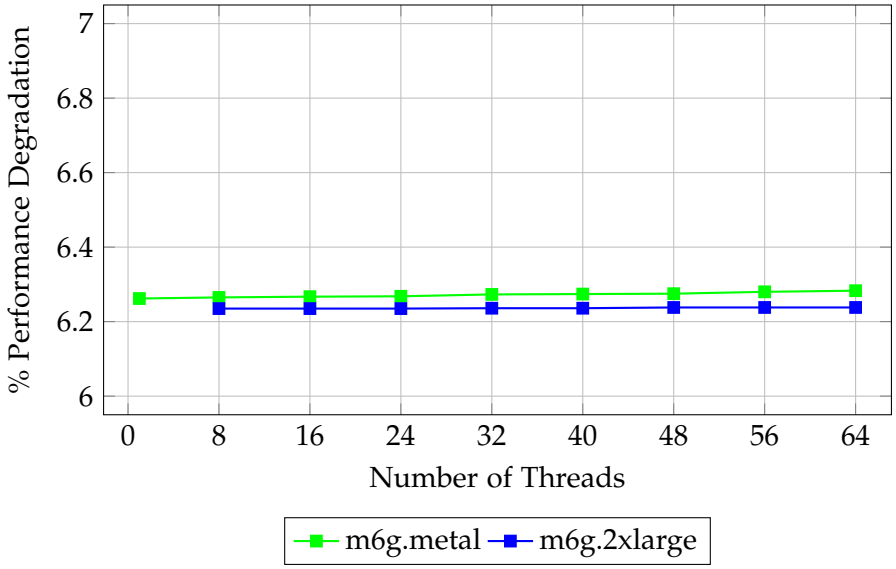


Figure 5.10: Effect of adding threads on the CPU performance using m6g.metal and the cpu_burn tool (V)

We notice a very little performance degradation on the total execution runtime, reaching a maximum of 0.33% at 64 threads. This result is expected as each new thread is assigned to an independent physical core. since the m6g.metal has 64 physical cores, the added threads before 64 should be assigned to an idle core and should practically have no effect on the other threads.

6 Network Resource Contention

DO NOT USE THIS DIRECTLY COPIED FROM AWS

- When it's specified that an instance has 10 Gbps of bandwidth, it means 10 Gbps of inbound and 10 Gbps of outbound traffic. However it still depends on
- Bandwidth for multi-flow traffic is limited to 50% of the available bandwidth for traffic that goes through an internet gateway or a local gateway for instances with 32 or more vCPUs, or 5 Gbps, whichever is larger. For instances with fewer than 32 vCPUs, bandwidth is limited to 5 Gbps.
- Bandwidth for single-flow traffic is limited to 5 Gbps when instances are not in the same cluster placement group.
- Typically, instances with 16 vCPUs or fewer (size 4xlarge and smaller) are documented as having "up to" a specified bandwidth; for example, "up to 10 Gbps". These instances have a baseline bandwidth. To meet additional demand, they can use a network I/O credit mechanism to burst beyond their baseline bandwidth. Instances can use burst bandwidth for a limited time, typically from 5 to 60 minutes, depending on the instance size.

In this section, we analyze the network contention that can occur between different residents of the same physical server. The available network bandwidth is a critical performance metric for applications, as it directly affects both the throughput and latency, therefore influencing the overall user experience. Unlike other resources such as CPU and RAM, which are clearly divided between tenants based on the instance type, network bandwidth is shared among the different co-tenants without a precise specification of the expected bandwidth per tenant. Typically, for instances with 16 vCPUs or less, AWS specifies the bandwidth upper bound. e.g., "Up to 10". However these instances have a baseline bandwidth. A network I/O credit mechanism is then implemented that allows these instances to use burst bandwidth for a short period of time, from 5 to 60 minutes, depending on the instance's type. The following table depicts all the specifications for the different instances types of the m5 family.

Model	vCPU	Maxiumum Burst Bandwidth (Gbps)	Baseline Bandwidth
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

For single flow traffic, the maximum burst bandwidth of 10 Gbps is only attainable when the the client and the server reside in the same cluster group. For instances who are not in the same cluster group, single flow traffic is limited to 5 Gbps. Bandwidth throttling for smaller instances takes at least 5 minutes to take effect, during which the instance has access to 10 Gbps burst bandwidth. We conduct our experiments in this time window to observe the impact of neighboring instances that are fully utilizing their bandwidth on the test node. It is particularly interesting to observe the extent of the network degradation in comparison to the baseline bandwidth for each instance type. Our following experiment is strcutured as follows: We use two m5 dedicated hosts, one that will host all the clients and the other will host all the servers. With each increment, we deploy a client node and a server node and execute the iPerf3 command on the client so that it's fully utilizing the bandwidth available to it, while continuously logging the results.

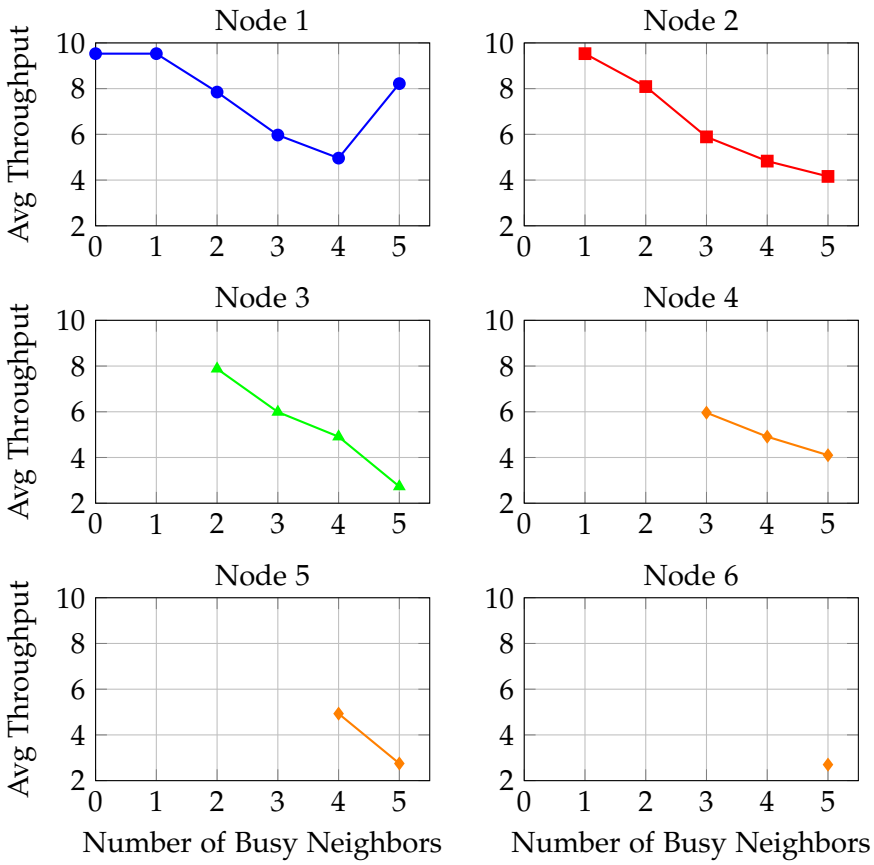


Figure 6.1: UDP Throughput of m5.4xlarge nodes when incrementally increasing the tenants

As expected, the first and second nodes had a throughput of 9.53 Gbps. The third tenant decreased the average throughput by almost 16.7% to 7.94 Gbps. The fourth neighbor introduced an average throughput decrease of 25%. The fifth neighbor introduced a degradation of 17%. At this point, we notice that all the nodes have similar throughput around 4.97 Gbps which represents the baseline bandwidth of the m5.4xlarge instance with practically no variation between them. The 6th Nodes introduced a 10.2% decrease. At this point, we witness a strong variation between the neighbors, with the first node reaching 8 Gbps and the 5th and 6th Node having a throughput of 2.7 Gbps. We also notice that the throughput can reach level lower than the baseline bandwidth of 5 Gbps specific to the m5.4xlarge. 2.7 Gbps is 46 % less than the baseline width of the m5.4xlarge instance of 5 Gbps. We repeat the experiment to see whether a pattern emerges. On average, with 6 tenants, the throughput is 4.11 Gbps which is 17.8 % less than the baseline width.

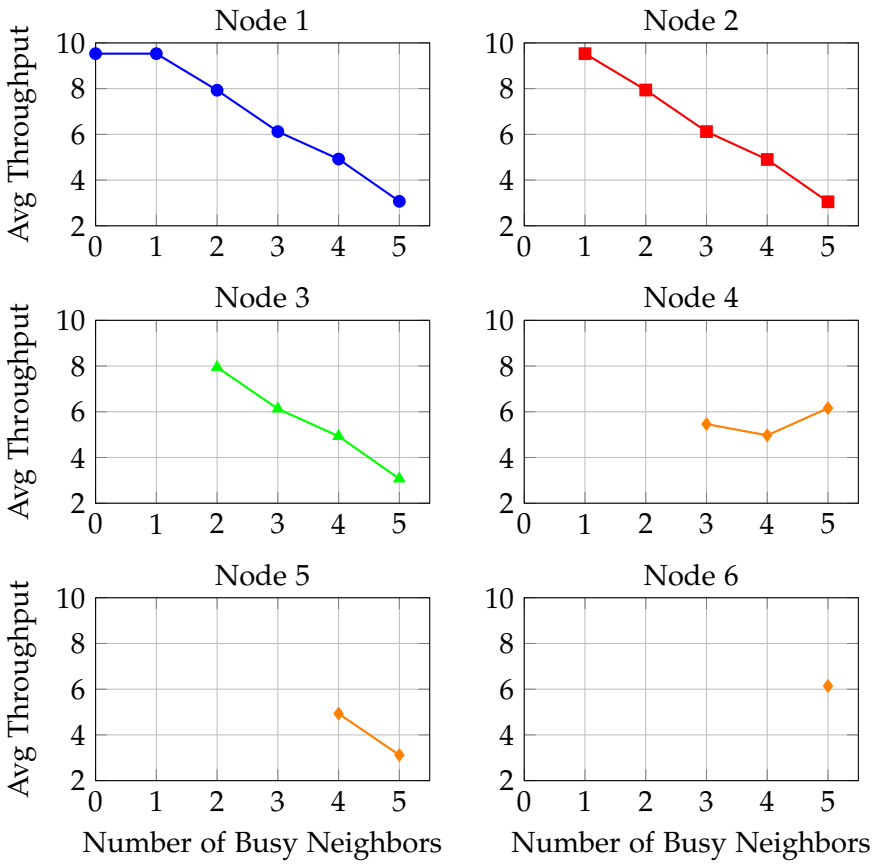


Figure 6.2: UDP Throughput of m5.4xlarge nodes when incrementally increasing the tenants

Up until the 5th tenant, we practically notice the same trend. with each tenant having a throughput of 4.9 Gbps. The variation happens at the 6th neighbor. However, this time we notice that the 4th and the 6th node have the advantage both with 6 Gbps bandwidth. We notice no clear pattern on how this preferation happens. The lowest throughput we witnessed here is 3.07 Gbps, which is 38.6% lower than the baseline bandwidth of the m5.4xlarge instance. Here as well, the average bandwidth at the end is 4.11 Gbps which is 17.8 % lower than the baseline width of 5 Gbps. In both experiments after introducing the 3rd tenant, we notice that the sum of the throughput off all the nodes is always around 24 Gbps. This is expected as the bandwidth of the m5.metal is 25 Gbps, which should be hard cap for the possible sum of the throughputs that are on the same dedicated host. For the xlarge, 2xlarge, 4xlarge types, the product of the possible number of tenants on the dedicated host multiplied by the baseline width is 30 Gbps, which is 16.7 % smaller than the possible bandwidth of 25 %. This

explains the average degradation of 17% we saw in the previous experiments and we should expect a similar behavior on the xlarge, 2xlarge types. For the large type howver the product is equal to 36 Gbps. 25 Gbps is 30% smaller than 36 Gbps. We should expect to see an average degradation of around 30% comparison if we repeat the experiment with using m5.large instances.

7 Conclusion

Abbreviations

SMT Simultaneous Multi-Threading

VMM Virtual Machine Monitor

IaaS Infrastructure-as-a-Service

IaC Infrastructure-as-Code

HCL HashiCorp Configuration Language

Bibliography

[1] 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.

[2] Amazon Web Services. *Amazon EC2 M5 Instances*. Accessed: 2025-06-12. 2024.

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

[4] 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.

[5] Intel Corporation. *What Is Hyper-Threading?* https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html. Accessed: 2025-05-23. 2025.

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

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

[8] N. Thorat. *AWS Graviton: Unlocking Performance and Cost Efficiency in the Cloud*. https://www.cloudyali.io/blogs/aws-graviton-performance-cost-efficiency. Accessed: 2025-05-23. 2024.

[9] Wikipedia contributors. *Hyper-threading — Wikipedia, The Free Encyclopedia*. Accessed: 2025-05-23. 2025.

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