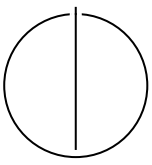
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

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

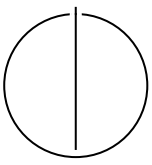
TECHNISCHE UNIVERSITÄT MÜNCHEN

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

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 the virtual machine providing users with limited transparency to 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 [15] [19]. Although virtualization technology has undergone major improvements in the past decades, contention can still occur due to other factors such as Simultaneous Multi-Threading (SMT), Network Interface Card queuing, and other system-level bottlenecks. To address the issue of unwanted resource contention, Amazon Web Services introduced dedicated hosts [5], which are physical servers that are fully dedicated to the customer, allowing users to deploy virtual machines with complete transparency over their placement. Another potential solution are the spread placement groups, which ensure that the VMs are placed on different physical servers. It can be particularly useful for highly parallel workloads, where the nodes have a similar workload nature (network- or CPU-intensive). However, this approach only mitigates internal resource contention and is limited to a small number of EC2 instances in the same availability zone [6].
In this paper, we leverage different benchmarking tools to assess the extent of the performance degradation that can occur because of VM co-location. We are particularly interested in quantifying the worst possible degradation that can happen by running identical resource-intensive benchmarks in parallel across VMs residing on the same dedicated host. We primarily utilized 5th and 6th general-purpose AWS EC2 VM instances that are built around the AWS Nitro system and run on top of the Nitro hypervisor. We analyze CPU contention across various CPU vendors namely Intel, Gravtion, and AMD. We examine whether and to what extent SMT is involved in this degradation. Furthermore, we investigate network resource contention, which is interesting since AWS does not provide exact specifications like other resources such as CPU and memory. We analyze 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, namely virtu-

alization and Simulatenous Multi Threading. In section 4, we explain the methodology of our experiments and introduce the different benchmarking tools that were adopted throughout the thesis. Section 5 analyzes CPU contention across 5th and 6th EC2 generations and contexualizes the results in relation to the CPU architecture. In Section 6, we examine network I/O contention and explore its manifestation across throughput and latency. Section 7 concludes our work and summarizes its most important findings.

2 Related Work (✓)

Han et. al [15] 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 Network throughput 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, which can be misleading. We will discuss this further in the network I/O section. 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 user to relocate VMs to have access to better performance with less contention.
Rehman et. al [22] 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 [19] 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 [13] developed an only software solution called Cuanta that predicts performance interference due to shared chip level resource 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 software of the hosting platform 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 emprical 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 [17] 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 [21]. 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 that's offered by cloud providers, as it provides various advantages [21]. 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 [8]. Most of the instructions that are executed by the virtual achines 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, that signals to the VMM to intervene and emulate the behavior of the instruction [8]. After the emulation is finished, the control is then given back to the guest OS, which is unaware of the underyling emulation. Several optimizations techniques have been introduced to reduce virtualization overhead, which will be briefly outlined later.
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 interface, storage drives, and input peripherals. Device models are required for this [8]. 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 [8]. 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 system.

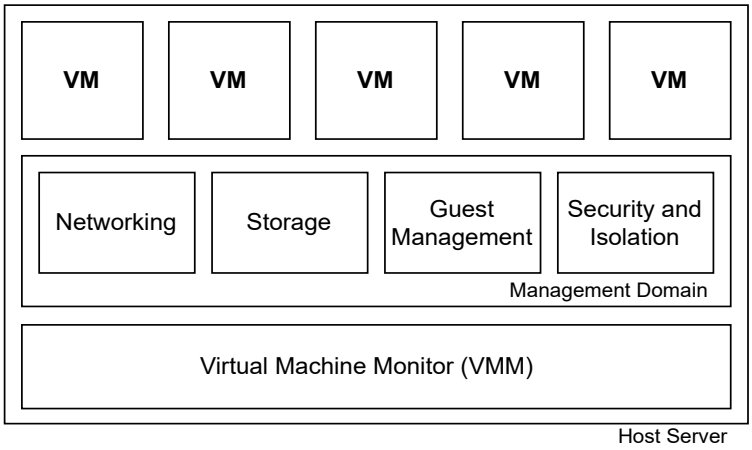


Figure 3.1: Architecture of traditional virtualization Solution

3.1.1 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 [14]. This is, however, very slow and can make the host apps run 2x to 10x slower [14].
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 downside of this approach is that it introduces additional complexity, as it requires modifications to the guest operating system [16].
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 adress translation space tasks [16]. 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) [8], 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 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 [8]. 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 [8].

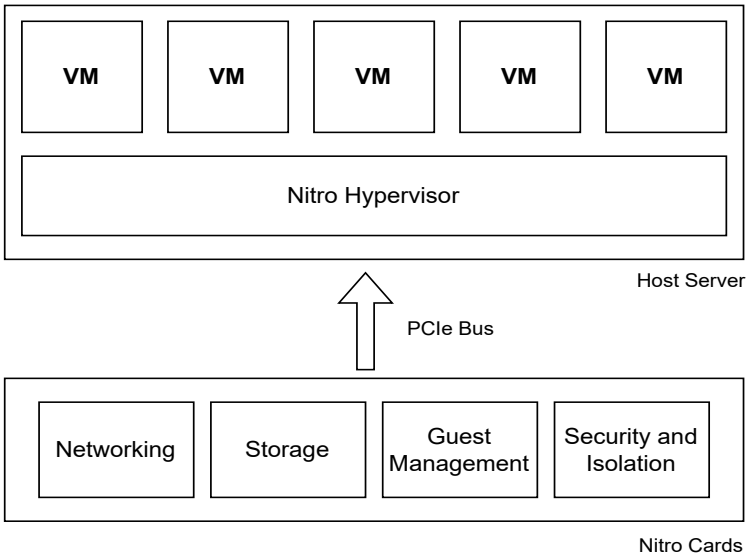


Figure 3.2: Architecture of Nitro System Virtualization

There are three main components in the AWS Nitro System [8].

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 [8]. 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. Example 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 main board. It's 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-restriced 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 [8].

The Nitro Hypervisor

The third component is the AWS Nitro Hypervisor, which has significantly fewer responsibility than traditional hypervisors, as most virtualization tasks are offloaded to the Nitro cards. It has three main functions. It's responsible for partitioning memory and CPU by using the virtualization commands provided by the processor. It's also in charge of assigning the virtual hardware interfaces provided by the Nitro cards to the Virtual Machines and also handles the machine management commands that come from the Nitro Controller (start, terminate, stop etc.) [8].

3.2 Simultaneous Multi-threading (✓)

Before we dive deeper into SMT, it's 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. To reach this, multiple techniques have been employed such as instruction pipelining, superscalar architecture and out-of-order execution [26]. 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 5th clock cycle, the CPU is simultaneously working on 5 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 [26]. 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 limit the usage of the available execution resources [26]. 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 inherently high. The wastages that occur on the processor can be categorized into two categories: Horizontal and vertical waste [26]. 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 [26].

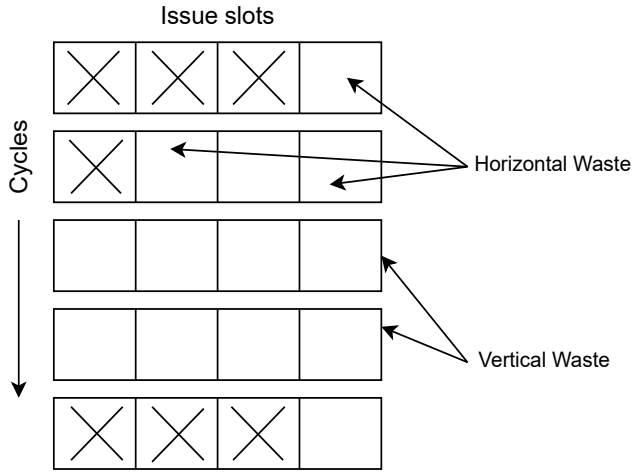


Figure 3.4: Vertical waste vs. horizontal waste

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 [26], effectively transforming each physical core into two (or more) "logical" cores. The idea is that instructions from different threads provide greater independency, which results in a better utilization of the core's execution resources [27]. 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 [27]. 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 [20].

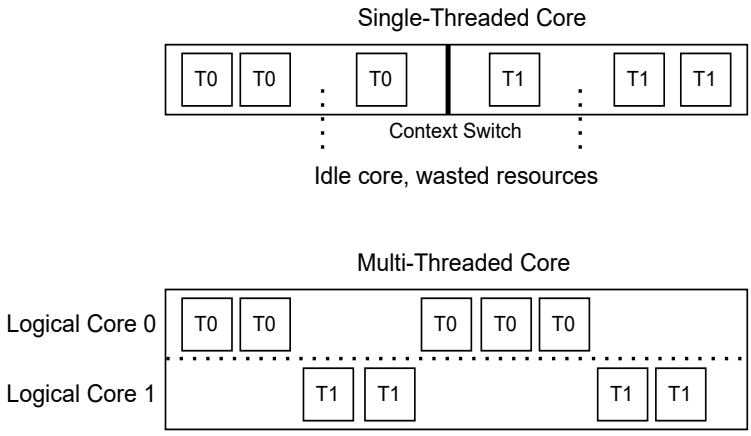


Figure 3.5: Single-Threaded Core vs. Multi-Threaded Core

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 [24]. 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 [24].

4 Testing Infrastructure

All tests were performed in the AWS us-east-1 Ohio region. We chose this region to minimize the costs, as it's considered to be one of the cheapest regions. The EC2 instances ran Ubuntu Server 24.04 LTS Linux. Throughout all the experiments, all the VMs were provisioned on dedicated hosts that are in the same availability zone and in the same Virtual Private Cloud. This is particularly important for network I/O experiments, as the network 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.
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), ensuring the automation and reproducibility of the benchmarks. Additionally, we used distexprunner [28], which is a tool written in python that helps write and run bash commands remotely across multiple nodes addressing them through 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 Python3 using mainly the re package [11] for parsing raw data and csv [10] for working with comma separated data. 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 included all the required software already pre-installed (python, distexprunner, benchmarking tools and monitoring utilities). The goal is to minimize the boot time of our instances and avoid configuration drift between runs.
These baseline conditions apply to both the network and CPU experiments. The specific experimental setups will be discussed in detail within the methodology section of each corresponding main chapter.

5 CPU Contention

5.1 Methodology

To generate CPU stress, we used sysbench [18], which is 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 [12]. The number of worker threads as well as 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 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 worker threads. It outputs the total wall-clock runtime that was needed for the execution of this job. We compiled the program using GCC with optimization level -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.

```
1 void* perform_work(void* arg) {
2     ThreadWork* work = (ThreadWork*)arg;
3     double x = 0.0;
4     for (long long i = 0; i < work->operations; ++i) {
5         x += i * 0.000001;
6     }
7     work->result = x;
8     return NULL;
9 }
```

Listing 5.1: Workload of the *cpu_burn* tool

This section investigates CPU contention on general-purpose instances from 5th and 6th EC2 generations, with different CPU architectures. We analyzed the potential

degradation on hosts that support Simultaneous Multi-Threading: namely m5, m6i and m6a hosts. We then considered the single-threaded AWS Graviton 2 processor used by the m6g dedicated host. 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. 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	m6g
Processor [9]	Intel Xeon Platinum 8175/ Intel Xeon Platinum 8280	Intel Xeon 8375C	AMD EPYC 7R13 Processor	AWS Graviton2
Hypervisor [3]	Nitro v2	Nitro v4	Nitro v4	Nitro v2
vCPUs [23]	96	128	192	64
Physical CPUs [23]	48	64	96	64
Clock speed (GHz) [7]	3.1	3.5	3.6	2.5
Hypervisor [2]	Nitro	Nitro	Nitro	Nitro
price/hour [23]	$5.069	$6.758	$9.124	$2.71
price/vCPU/hr	$0.053	$0.053	$0.048	$0.042

Table 5.1: KPIs for AWS Dedicated Host families m5, m6i, m6a, and m6g.

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.

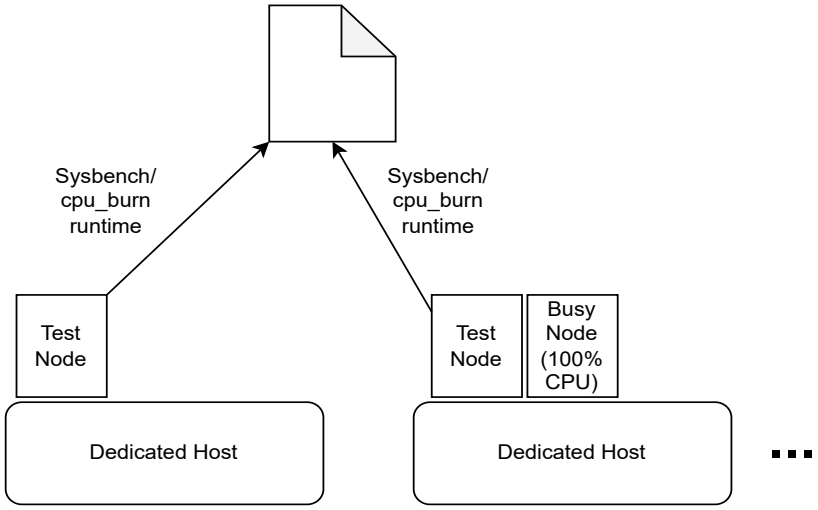


Figure 5.1: CPU contention experiment

For comparison, 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 the user with access to all the physical CPUs. The behavior of the metal instance will not be analyzed in depth, as it is out of the scope of the thesis. It is included solely to support a better understanding of CPU contention.

5.2 Contention under SMT

5.2.1 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 [4]. 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 5.2: m5 Instance Specifications [4]

The m5 dedicated host used the 2nd-generation Intel CPU in all our experiments. We repeated the benchmarks 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

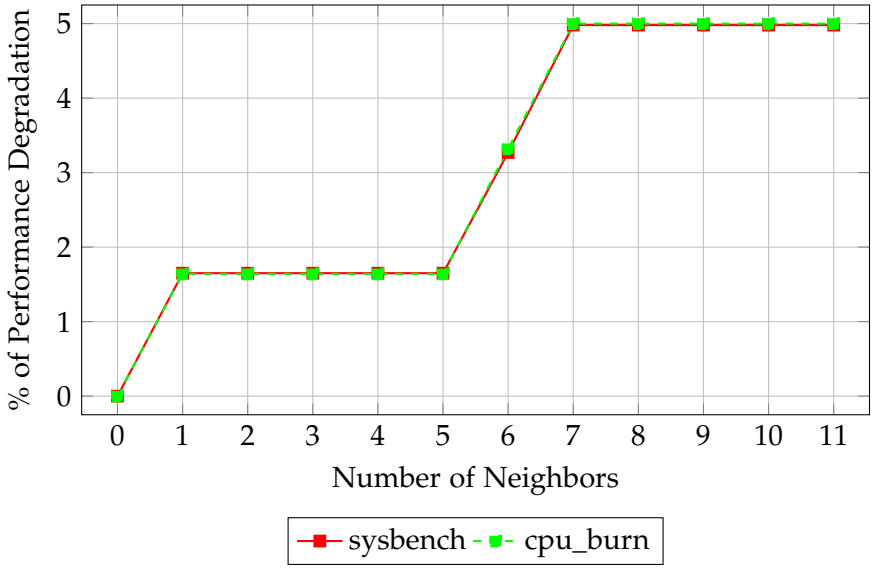


Figure 5.2: Effect of adding busy neighbors on the CPU speed of the sysbench and cpu_burn command on the test node using m5.2xlarge instances

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. Potential reasons could be physical core co-location between the neighbors and the test node or hypervisor overhead. The latter is very unlikely, as the Nitro system, along with hardware-assisted virtualization should introduce a very small overhead. We repeat the same experiment but adding idle VMs. Figure 5.3 summarizes the results.

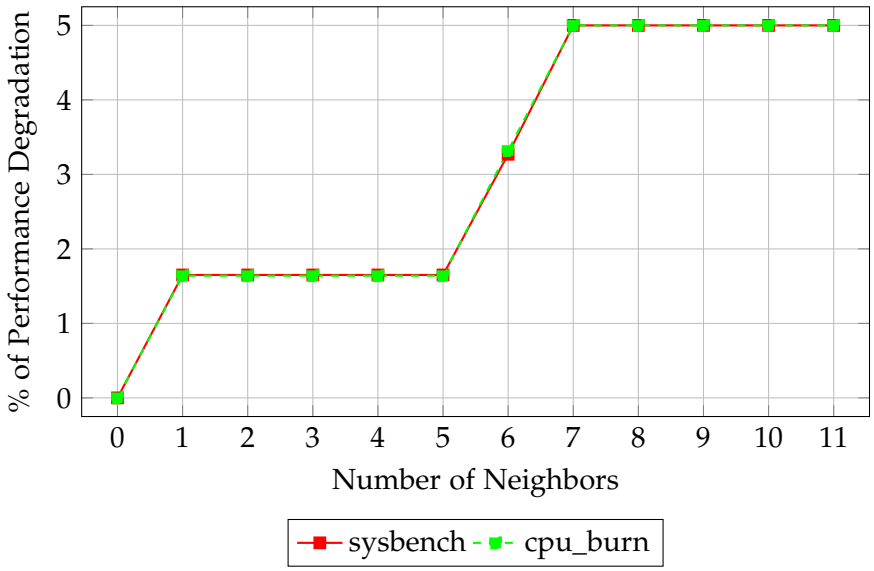


Figure 5.3: Effect of adding idle neighbors on the CPU speed of the sysbench and cpu_burn command on the test node using m5.2xlarge hosts

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.

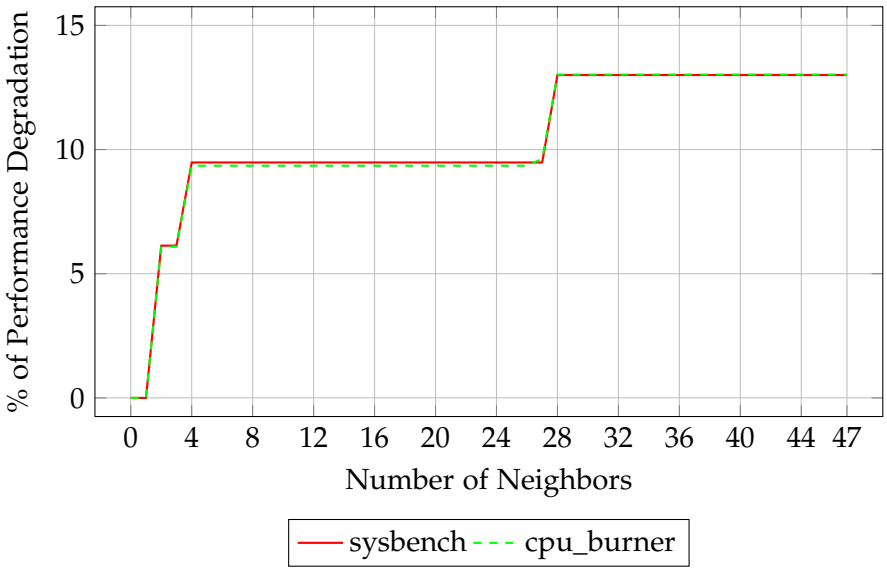


Figure 5.4: Effect of adding busy/idle neighbors on the CPU speed of the sysbench and cpu_burn command on the test node using m5.large instances

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.

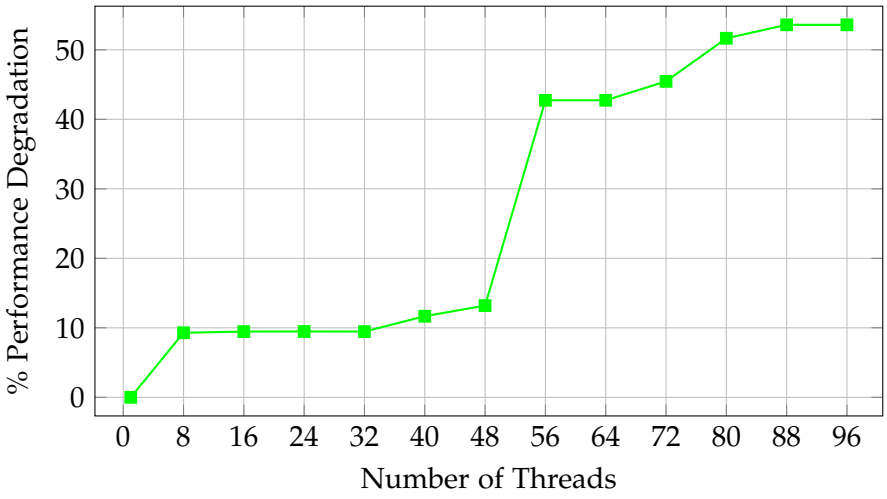


Figure 5.5: Runtime impact of incrementally increasing threads in the *cpu_burn* command on an m5.metal instance

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 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. 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.

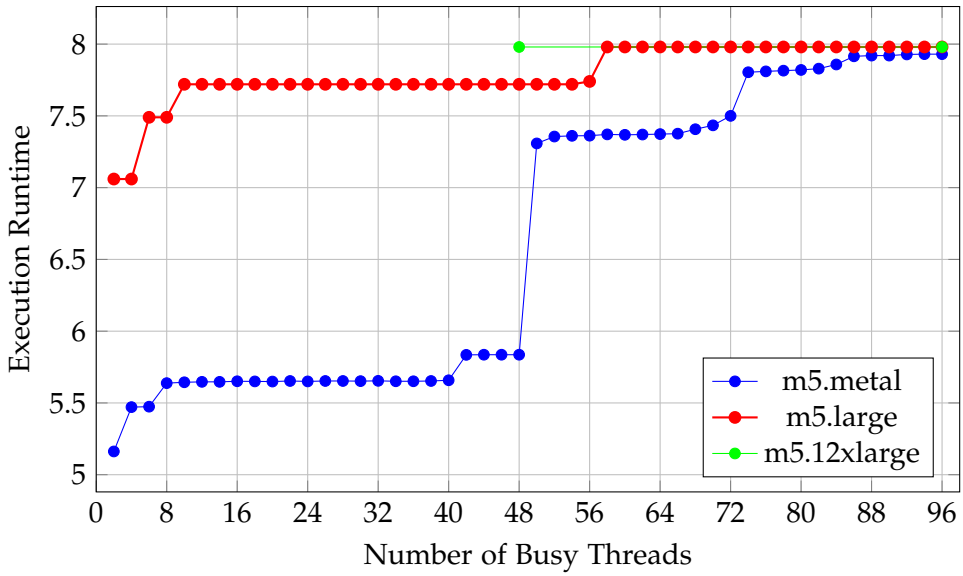


Figure 5.6: Performance of the test node (while incrementally adding busy neighbors) in comparison to running the threads natively on m5.metal

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 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 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. However, it is highly unfavorable when the user has access to a dedicated host, as they are unable to fully utilize the available idle CPU resources of the physical machine, despite paying for all of them.
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.

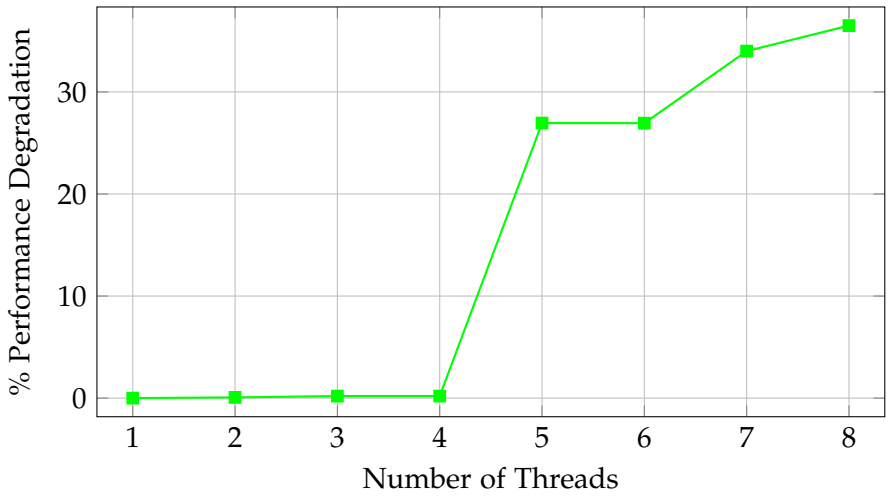


Figure 5.7: Effect of adding busy threads on the CPU performance of a m5.2xlarge instance using *cpu_burn*

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%.

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 5.6. 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 5.6.

5.2.2 m6i family

It is worth exploring whether, and to what extent, the behavior we witnessed for the m5 family is present on the 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 5.7 shows.
These results can be misleading in suggesting that the performance degradation for the m6i family. However, 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 in comparison to the other types. 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.

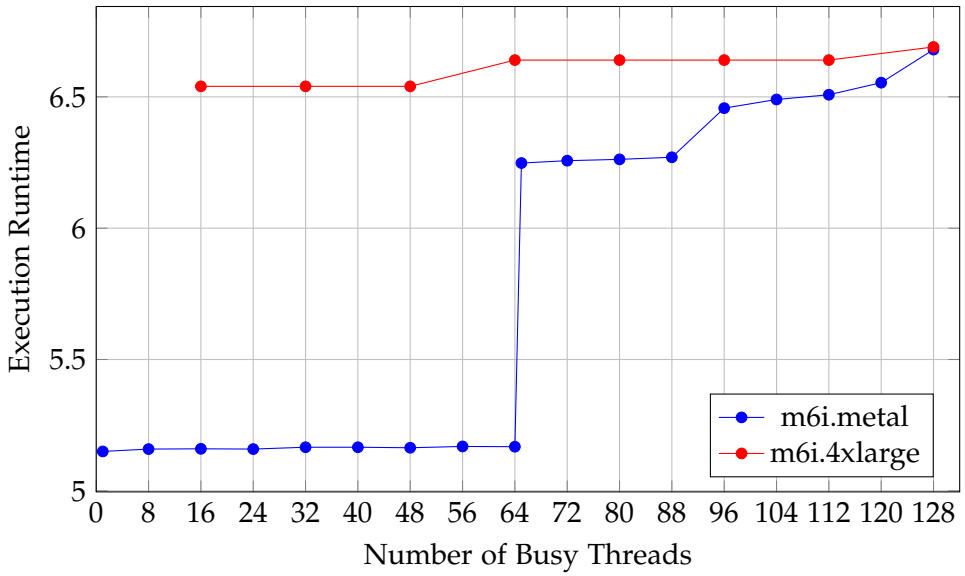


Figure 5.8: Performance of the test node in comparison to running the threads natively on m6i.metal

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 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 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.

We investigate whether this behavior is also present on processors from other vendors that support SMT, such as certain AMD models.

5.2.3 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.

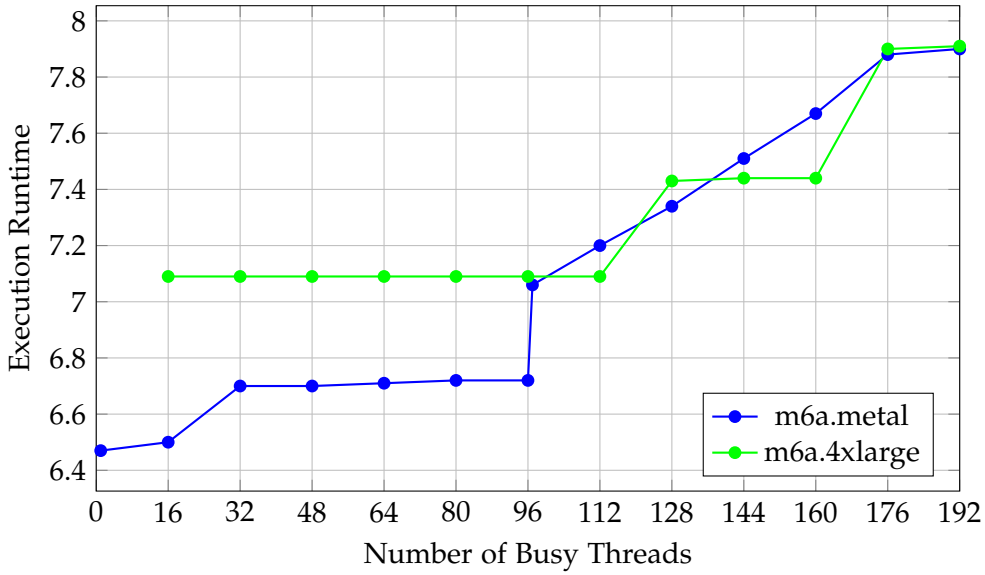


Figure 5.9: Performance of the test node in comparison to running the threads natively on m5.metal

We notice a pattern similiar to that of the Intel processors. However, the overall performance degradation is less important and 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 for the Intel processors.

5.3 Contention under Single Threaded Core Processors

5.3.1 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 5.6 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 5.6: vCPU and RAM specifications for m6g instance types

For our first experiment we used m6g.2xlarge nodes, of which the dedicated host can provision 8 instances. The results can be seen in figure 5.10.

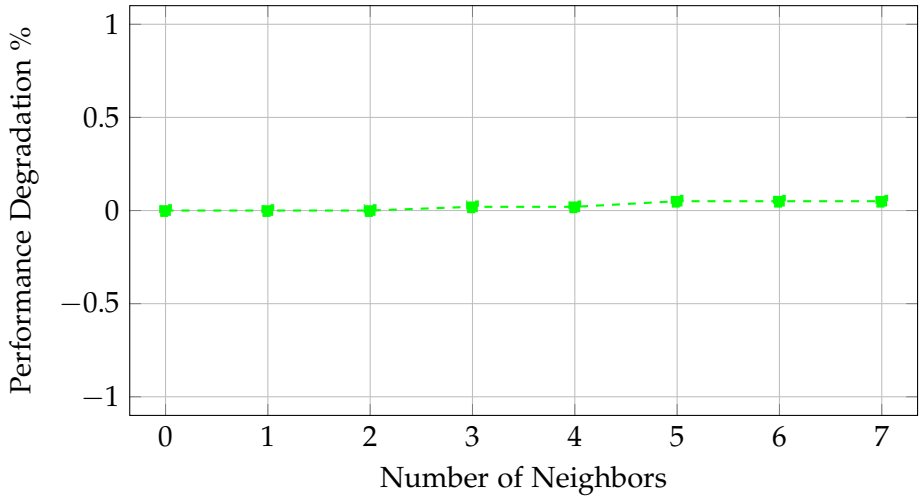


Figure 5.10: Effect of adding busy neighbors on the CPU speed of the *cpu_burn* command on the test node using m6g.2xlarge instances

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. Table 5.7 captures the final performance degradation for different instances types.

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

Table 5.7: Maximum achievable performance degradation on our test node across various m6g instance types

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.
Figure 5.11 depicts the effect of adding busy m6g.2xlarge neighbors on our test node in comparison to the runtime of running the threads natively on the m6g.metal instance.

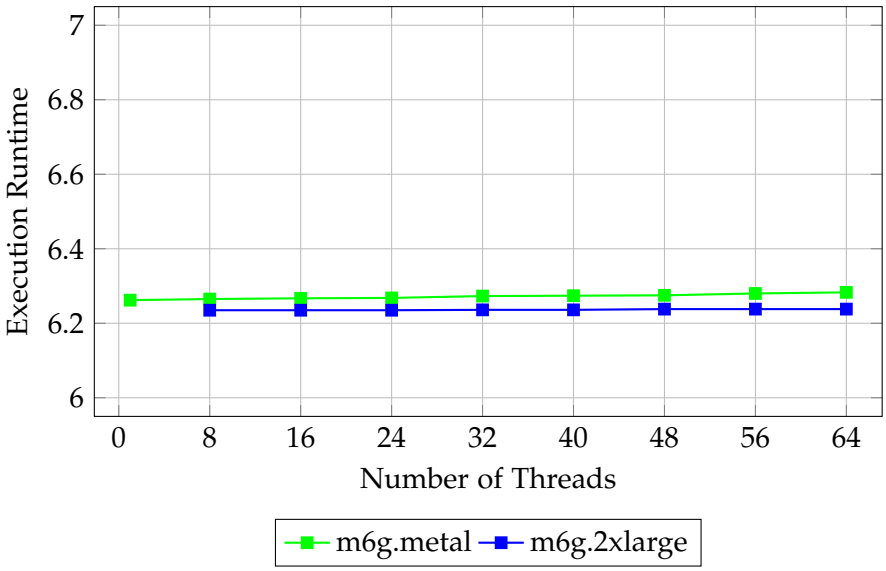


Figure 5.11: Effect of adding threads on the CPU performance using m6g.metal and the cpu_burn tool

For the m6g metal, we observe 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, and have practically no effect on each other. We noticed that the m6g.2xlarge even delivered a slightly better performance throughout the experiment, though the difference was minimal and likely insignificant.

5.4 Discussion

The m5 instances are prone to CPU contention, with the percentage of performance degradation ranging from 3.25% (4xlarge) to 13% (large) (Table 5.3). Particularly for this family, experiments show a degradation caused by adding idle neighbors, on par with the degradation from adding busy neighbors. In their work, Han et al. [15] observed this behavior and were able to improve sysbench performance by 20.81% on m5d.large instance by shutting down idle VMs. This behavior seems unique to the m5 family as they were not able to replicate it on c3 and c4 families. In our work as well, m6i and m6a families did not exhibit this behavior, as idle VMs caused sub 0.05% degradations, which are practically insignificant. Han et al. [15] argue that this performance degradation is caused by context switching overhead that's performed by the KVM (Nitro) scheduler. We think that this is very unlikely, since the Nitro System provides a near bare-metal performance. Experiments with the m6g family, that features the same hypervisor as the m5 dedicated host, support this conclusion (see Table 5.7). m6i instances revealed a minimal degrdation effect, with performance degradation below 2.5% across all instance types (see Table 5.4). However, this was not the case for the m6a instances that exhibited a degradation close to 10% across all tested instance types.

For dedicated hosts that support SMT, we constantly observe that the performance of the first deployed VM is significantly worse than the performance of running the same number of busy threads directly on the metal offering. This intital difference reached 36% on m5, 27% on m6i, and 9.3% on m6a using large instances. A plausible explanation is that instances with n vCPUs from these families have access to only $n/2$ physical cores, which is supported by the results in figure 5.7.

The metal instance across these 3 families demonstrated a significant degradation when incrementally adding busy threads directly on the physical CPUs: 53% on m5.metal, 31.85% on m6i.metal, and 23.5% on m6a%. This is mainly but not only caused by core co-location of the different threads. The performance degradation, we witnessed for the test VM in the experiments is basically the VM following the trend of the performance degradation of the underlying CPU, but with a worse start as its vCPUs share the physical cores. The degradation reflects the performance, i.e., it's a characteristic of the underlying processor. Virtualization overhead does not contribute to this degradation. This is supported, by the fact that the test node runtime and the metal instance runtime always converge to the same value when the the metal instance and the dedicated host are at full capacity.

For the m6g dedicated host, we witnessed almost no CPU contention, when adding busy neighbors, with all values below 0.05%. This was expected as the AWS Graviton processor provides better vCPU isolation with each logical core mapped to a physical

core. The neighbors still share Last Level Cache and memory system, but that does not seem to have an effect as our benchmarks are CPU-bound.

6 Network I/O Contention

6.1 Throughput

6.2 Methodology

6.2.1 Latency

For latency benchmarking, we employed both *iPerf3* and *sockperf* [25]. Sockperf is a network benchmarking utility capable of measuring the latency of packets with a sub-nanosecond resolution. It introduces minimal overhead by leveraging the Time Stamp Counter (TSC) registers that count the number of CPU cycles for measuring latency [25].
Sockperf requires a client-server setup as well: 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 consists of the following steps: Similar to the throughput experiment, we deploy two dedicated hosts, one hosting all clients and one hosting all servers. Initially, only the client test node and its corresponding server are created. We measure the latency between them and record it as a baseline value. After that, we deploy all the remaining clients and incrementally increase their aggregate throughput usage in 1 Gbit/s steps (or smaller as we notice the presence of latency degrdation). This is possible through an iPerf3 functionality that allows the specification of an exact throughput value for the client, allowing us to control the aggregated throughput of the neighbors. The test client node and its corresponding server are the only nodes using the sockperf tool. 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 is minimized, allowing for a better identification of latency degradation.
The overall architecture of the latency experiment is similar to Figure **??**. The two key

differences are: (i) the test pair (client and server) used sockperf instead of iPerf3, and (ii) The neighboring nodes did not immediately saturate the available bandwidth, but instead progessively increased their throughput.

6.3 Throughput Contention

6.3.1 m5 family

Unlike other resources such as CPU and RAM, which are statically divided between the different tenants based on instance type, network bandwidth is shared among the different co-tenants without a fix 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 Gbps" [1]. However these instances still have a baseline bandwidth. A network I/O credit mechanism is then employed that allows the instance to use burst bandwidth for a short period of time, ranging from 5 to 60 minutes, depending on the instance type [1].
In this section, we'll analyze throughput contention on instances belonging to the m5 family. Key performance metrics 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) [9]	Baseline BW [9]
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)

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 [1]. The clients and the servers can't be in the same cluster groups since they are in different dedicated hosts. This means that the single-flow traffic between the client and the servers is limited to 5 Gbps. To bypass this restriction, we create two client connections, that can be achieved using the P option in the iPerf3 tool.
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. It is particularly interesting to observe the extent of the network degradation in comparison to the baseline bandwidth for each instance size. The first experiment features the m5.4xlarge instance type, of which the dedicated host can accomoadate 6. To provide a better visualization of the results, we present each node in an independent graph.

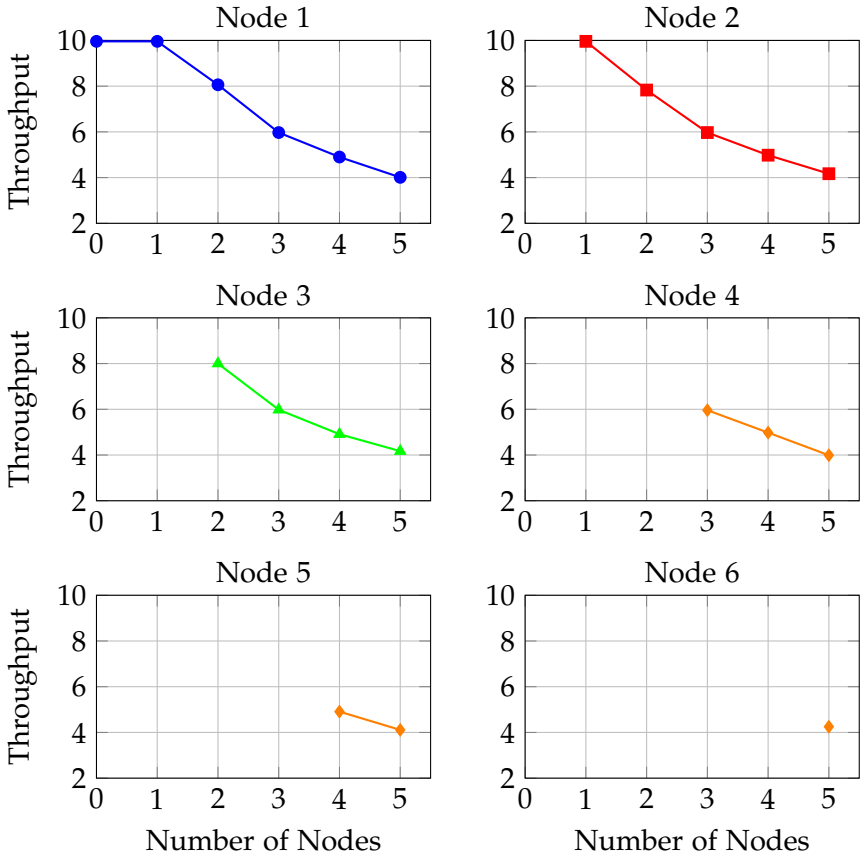


Figure 6.1: UDP Throughput of m5.4xlarge nodes when incrementally increasing the tenants

As expected, the first and the second clients, when alone on the dedicated host had ac-

cess to a burst bandwidth of 9.96 Gbps. The third tenant caused the average throughput to drop to 7.96 Gbps, while the addition of a fourth neighbor further reduced it to 5.96 Gbps. We practically notice no variation between the different nodes. The fifth neighbor reduced the throughput of all the co-tenants to practically the baseline width of the m5.4xlarge (5 Gbps) with an average throughput of 4.94 Gbps. The 6th node introduced the first significant decrease under the baseline width to an average of 4.12 Gbps, which is 17.6% less than the baseline width. Starting from the the 3rd tenant, the sum of the throughput across all the nodes is always around 24.7 Gbps. This was expected as the bandwidth of the m5.metal is 25 Gbps, which should represent 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 the instances on the dedicated host multiplied by the baseline width of the respective instance type (5 Gbps) is 30 Gbps, which is 16.7% smaller than the possible bandwidth of 25 Gbps. This explains the average degradation of 17% we saw 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 $48 \times 0.75 = 36$ Gbps. 25 Gbps is 30% smaller than 36 Gbps. We should expect to see an average degradation of around 30% at full capacity if we repeat the experiment using m5.large instances. We verify this assumption in the next experiment. Since the dedicated host can host 48 of m5.large instances, we can't individually plot the graph of each instance. We used a plotbox graph to display the distribution of the throughputs at the different levels. We also used a step size of 8 neighbors, to be able to present the results in one graph.

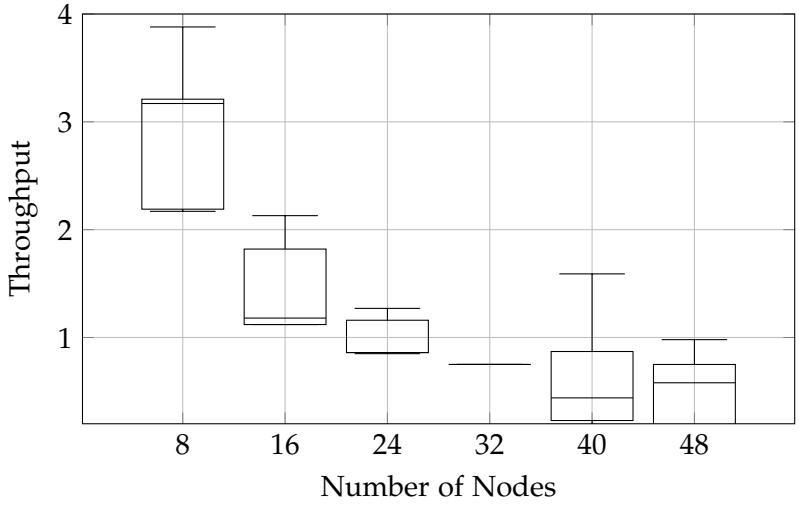


Figure 6.2: Throughput (UDP) of m5.large nodes when incrementally increasing the number tenants

The first node has access to a throughput of 9.96 Gbps, which represents the maximum burst bandwidth. At 8 neighbors, the average throughput drops 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 interestingly notice zero variation between the different co-tenants. At 40 nodes, the average decreases to 0.614 Gbps and further to 0.51 Gbps at 48 nodes. At full capacity, The average throughput (0.51 Gbps) is 30.7% lower than the baseline bandwidth of the m5.large instance (0.75 Gbps). This more significant performance degradation close to 30% was expected as hypothesized earlier. In this experiment, we also observe an important performance variation between the different nodes compared to the previous experiment.

6.4 Latency

6.4.1 m5 family

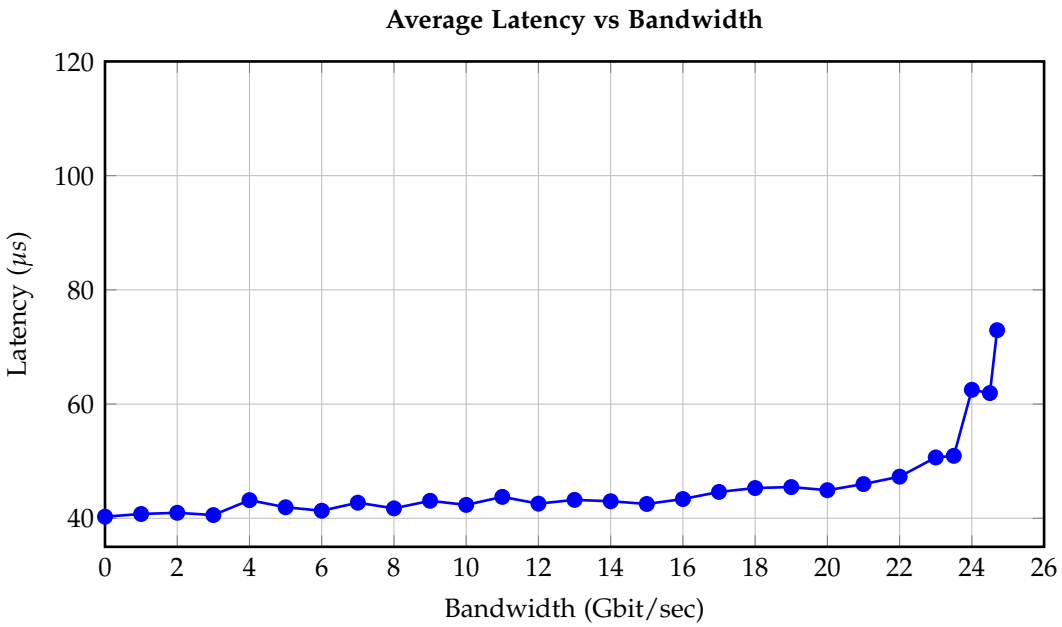


Figure 6.3: Average Latency vs Bandwidth

6.4.2 c7g family

6.5 Discussion

7 Conclusion

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*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html#instance-hypervisor-type. Accessed: 2025-08-02.

[3] 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.

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

[5] Amazon Web Services, Inc. *Amazon EC2 Dedicated Hosts*. Accessed: 2025-07-26.

[6] Amazon Web Services, Inc. *Placement groups for your Amazon EC2 instances*. Accessed: 2025-07-26.

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

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

[9] T. DIS. *CloudSpecs*. urlhttps://tum-dis.github.io/cloudspecs/. Accessed: 2025-08-02.

[10] 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.

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

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

[13] 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.

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

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[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.