

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI INGEGNERIA DELL'ENERGIA ELETTRICA E
DELL'INFORMAZIONE “GUGLIELMO MARCONI”

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA E
TELECOMUNICAZIONI PER L'ENERGIA

Non-invasive network analytics estimation for containerized network functions

Master's Thesis in
Telecommunications Network Management LM

Supervisor

Prof. Davide Borsatti

Submitted by

Agostino Fiumana

Co-supervisors

Prof. Leonardo Linguaglossa

Prof. Walter Cerroni

II Session, September
Academic Year 2025/2026

Let a , b , and c be the side lengths of a triangle, and let

$$s = \frac{a + b + c}{2}$$

be its semiperimeter. Then its area is given by

$$A = \sqrt{s(s-a)(s-b)(s-c)}.$$

Heron's formula

Contents

Abstract	5
1 Introduction	7
2 Shared Hardware Resources	9
2.1 CPU Execution	9
2.1.1 Computation	11
2.1.2 Caches and I/O	12
2.1.3 Main Memory	14
2.2 Frequency and Power	15
2.3 NUMA Nodes	15
2.4 CPU Time and Scheduling	16
2.5 Characterising Contention	16
3 Cloud Deployments	19
3.1 Virtualisation	20
3.2 Control Groups	23
3.3 Orchestration	24
3.4 Cloud to Hardware Interaction	25
3.4.1 Address Translation	26
3.4.2 Interception	26
3.4.3 The I/O Path	27
3.5 Telemetry in virtualised Environments	29
4 Observing Performance Under Contention	31
4.1 Network Telemetry	31
4.2 The Observer Effect	33
4.3 Inferring Performance from Hardware Events	34
4.4 Performance Monitoring Units	35
4.4.1 Counter Affinity	37

Contents

4.4.2	Counters Under Virtualization	39
5	Setup and Methodology	41
5.1	Contention Experiment Topology	43
5.2	Workload Progression	45
5.3	Experimental Setup	46
5.4	KPI Measurement	47
5.5	Obtained Datasets	48
6	Experiments	49
6.1	Cross Topology Analysis	49
6.2	Variable Frequency Effects	51
6.3	Cross Workloads Analysis	52
6.4	Findings	54
7	Leveraging Data	57
7.1	Feature Construction and Validation	57
7.2	Per-Second KPI Estimation	58
7.3	Aggressor Localization	61
7.4	Regime Classification	62
7.5	Estimation Layer	63
8	Conclusions	65
	Acknowledgements	67
	Bibliography	69

Abstract

Il passaggio da funzioni di rete implementate su hardware dedicato alla controparte realizzata tramite software ha reso le prestazioni dei sistemi di rete più ostiche da prevedere poiché diverse funzioni che condividono lo stesso hardware possono deteriorare le performance sia per la saturazione di risorse che per la mutua interferenza tra processi. Alle velocità di traffico odierne le misurazioni di velocità e latenza per pacchetto, diventano impraticabili; e l'operatore che ospita funzioni di terzi sulle proprie infrastrutture deve garantire la qualità del servizio senza conoscere i processi in utilizzo dagli utenti. Questa tesi esplora un metodo per osservare l'infrastruttura a basso impatto, nonostante l'alto throughput, l'interferenza dei processi e la difficoltà nel conoscere il layer applicativo in un sistema di rete. Sfruttando i contatori hardware delle Performance Monitoring Unit, si riconduce il comportamento microarchitetturale di una macchina ai Key Performance Indicators di rete nel caso di funzioni containerizzate. Funzioni reali sono misurate durante la competizione per le risorse, variando la collocazione dei processi interferenti nella topologia del sistema hardware, per stabilire come la sola telemetria hardware consenta di stimare la latenza percepita dal client e di distinguere il processo di interesse dalle sorgenti di interferenza. I risultati mostrano che dai soli contatori del processo osservato è possibile, tramite modelli di Machine Learning, ricostruire la latenza al 99° percentile secondo per secondo su esecuzioni mai viste in addestramento con $R^2 = 0,87$, che aumenta a 0,96 su un unico modello che copre payload di dimensioni diverse, individuare la collocazione dell'interferente fra cinque posizioni nell'hardware con accuratezza dell'83%, e riconoscere il regime di lavoro della funzione anche su un workload assente dall'addestramento senza alcuna visibilità sul software degli utenti.

1 Introduction

Moving from hardware based middleboxes to software defined functions exposes the network to new challenges. Over the years, providers strived towards leveling up traffic processing capabilities while embracing the dynamic nature of virtualized network functions. What stems from the heterogeneous nature of such implementations is a much more unpredictable behaviour due to the contention of multiple processing relying on shared hardware resources. Hence, predicting performance in virtual environments adds a fundamental complication, even though resources may not be necessarily saturated processes can interfere with each other. To better evaluate how these novel interactions can undermine performance, telemetry tools and process profiling are essential. As scaling capabilities of the network evolve so does the processable number of packets for software defined functions: Google's Maglev load balancer reaches line rate on a 10 Gbps link with minimum-size packets (14.88 Mpps) [1]. At such high rates traditional per-packet measurement approaches prove costly and impractical. This thesis aims to explore novel solutions that enable the estimation of network Key Performance Indicators (KPIs) in high-traffic scenarios and to distinguish relevant network processes from sources of interference on the same hardware. In order to achieve this the research work conducted examines the correlation between hardware related information and high-level behaviour of containerized functions with the intent of bridging the two using machine learning techniques. Network operators nowadays provide their infrastructure to many users running custom functions that need high computational power, in most cases operators cannot access the software level of a function but are still required to provide high Quality of Service standards, using information about the hardware to infer on the software stack is a way for operators to get intel on their infrastructure without touching the user space reserved to their clients. This asymmetry between provider and user is one of the many reasons why hardware-based solutions can offer benefits, enabling smarter resource allocation and scheduling of processes over huge infrastructures. The thesis background introduces both the operation of hardware resources and how multiple processes can compete for them, as well as the entire

1 Introduction

virtualisation and cloud computing layer that affects network functions. Finally, it describes how the behaviour of the lower layers can be observed using Performance Monitoring Units (PMUs) and how these apply to containerised network functions. The work concerning research is presented with an introductory section describing the methodology and the software used to conduct the experiments, followed by an analysis of the results obtained and how they can be utilised to gain insights into a system's performance.

2 Shared Hardware Resources

Performance interference between colocated workloads is a hardware phenomenon observed through software. When processes share hardware resources they interfere with each other slowing down the execution of one another. There's many different scenarios where functions may compete for hardware resources, based both on the location and the resources the function exploits most. This section covers all the shared hardware going from least to most shared among processes.

2.1 CPU Execution

A central processing unit (CPU) is the component that executes a program's instructions, built from a datapath that performs the operations and a control unit that sequences them. Its execution time for a given program follows from three quantities alone, the number of instructions executed, the average number of cycles each of them takes, and the duration of a cycle [2]. Modern CPUs are strongly optimised to keep their pipeline busy, applying techniques such as hardware prefetching and predictive speculation in order to not waste any operation [3]. This effort translates in a rich architecture with many interacting elements. From a macroscopic point of view, the pipeline of modern CPUs has two main portions: a frontend and a backend. The frontend is responsible for fetching instructions from memory and translating them into micro-operations (μ ops) which are then fed to the backend portion which is responsible to schedule, execute and retire these μ ops in the original program order. In order to keep the execution balanced these μ ops are typically buffered in "ready- μ ops-queue" once ready to be retired by the backend [3]. Figure 2.1 shows a general Intel CPU architecture.

In networking a server presents itself to an operator as a pool of logical CPUs, and a tenant is allocated some number of them. That allocation is not a partition of the machine since a logical CPU is not isolated from other tenants as almost every structure that determines how fast it runs is shared with a specific and enumerable set of neighbours.

2 Shared Hardware Resources

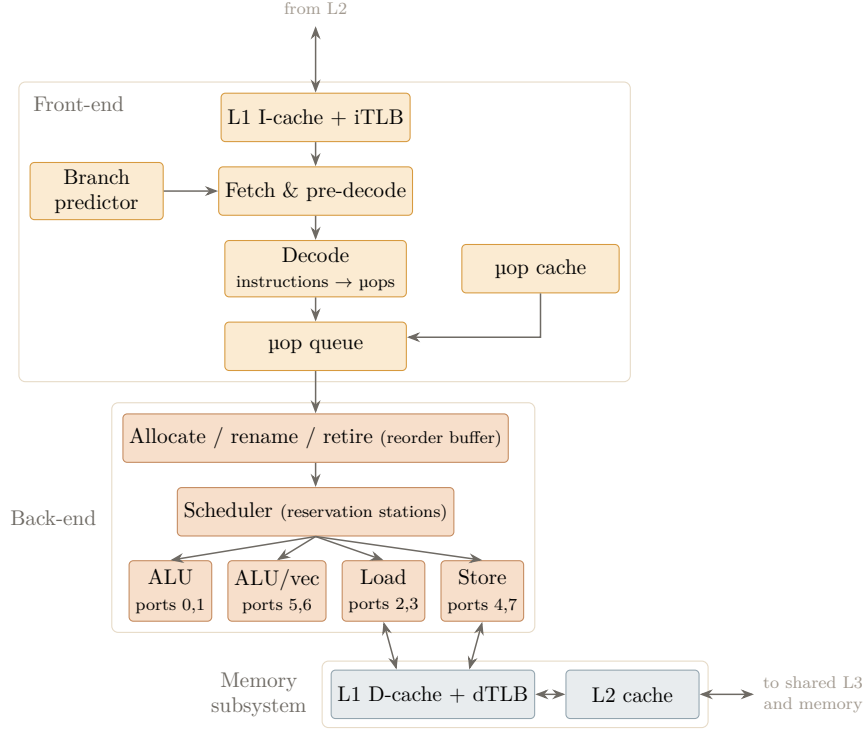


Figure 2.1: CPU pipeline execution

For a process running on one logical CPU, three questions define its exposure to colocated work: what resources are exclusively held, which are shared and with whom.

Shared resources may have different levels of locality, as they may be contested by an increasingly large number of tenants depending on the hardware’s topology. Figure 2.2 presents an example topology.

As shown the server features two sockets both containing a NUMA node with local caches associated, each core is clearly divided into two logical cores and owns two caches: L1 and L2. Socket 0 is also connected to Network Interfaces via PCI. Thus two processes sharing a logical core will share every single resource built on top while processes in different NUMA nodes will share only Main Memory access which is common to every socket.

Given this architecture, three main shared components are identified as affecting the changes in the CPU execution, namely: computation, cache accesses and memory operations [4].

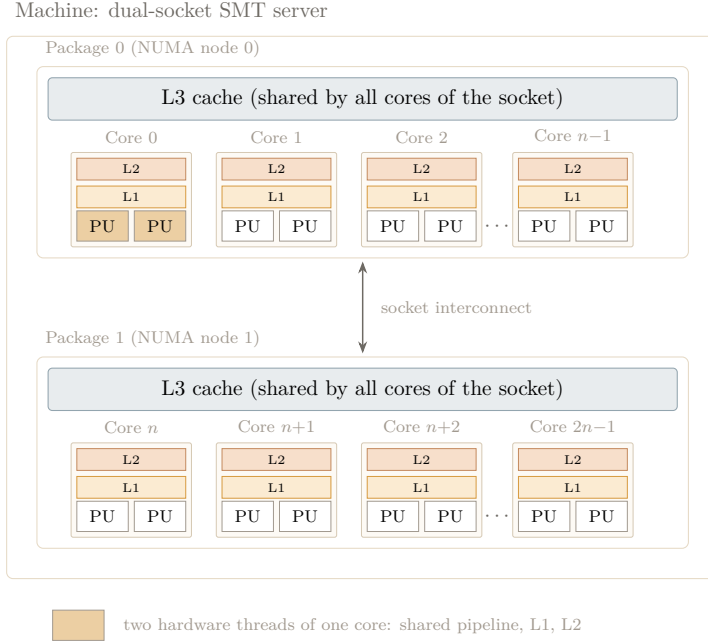


Figure 2.2: Example server topology

2.1.1 Computation

The narrowest shared resources are inside the core itself. A modern core is not a single sequential unit but a set of structures that a thread occupies competitively: a front end that fetches and decodes instructions, a reorder buffer (ROB) that holds instructions in flight, reservation stations that hold operations awaiting operands, and a set of execution ports, each serving a particular mix of operation types. Branch prediction structures and the μ -operations cache are shared as well.

Interference in this region is caused by structural occupancy: instructions from one thread hold ROB entries that another thread cannot use, or micro-operations from one thread are dispatched to a port in a cycle in which another thread also had work ready for that port. Thus performance drops are due to arbitration, not eviction.

ROB saturation produces a pattern in which instructions accumulate in flight without retiring, so that stall cycles rise while the number of cycles executed stays flat. Branch prediction behaves differently again: sharing predictor state between threads degrades accuracy for both, but a thread that executes less work also mispredicts less often, so recovery-cycle counts can fall while performance degrades.

In order to avoid stall cycles modern CPUs use simultaneous multithreading (SMT). While SMT is active the CPU exposes two execution contexts per physical

core, thus one physical core works like two logical cores that can handle different software threads and can work through tasks more efficiently than a single-threaded core, by leveraging the idle time when the core would otherwise be waiting for other tasks to complete [5]. The operating system schedules them independently and presents them identically, but they are not independent hardware. Only architectural register state is duplicated. The two threads share the execution ports and the ROB described above, together with the L1 instruction and data caches, the unified L2, the translation lookaside buffers, the micro-op cache, and branch prediction state.

Some structures are statically partitioned while both threads are active, so that each receives a fixed fraction regardless of demand. Others are competitively shared, so that a thread with higher demand displaces the other. Cache capacity falls into the second category: a sibling thread with a large working set evicts the main thread data from L1 and L2.

The characteristic signature of SMT contention follows from this. Both threads continue to execute and neither is descheduled. What changes is how much work each retires per cycle, because they compete for issue slots on every cycle in which both have instructions ready. The effect therefore appears in instructions per cycle (IPC) and in private cache hit rates, and not primarily in the number of cycles.

2.1.2 Caches and I/O

Cache memories are small, high-speed buffer memories used to hold temporarily those portions of the contents of main memory which are currently in use. Information located in cache memory may be accessed in much less time than that located in main memory. Thus, a CPU with a cache memory needs to spend far less time waiting for instructions and operands to be fetched or stored [6]. Modern architectures feature two local caches attached to the core itself: L1 and L2 caches, smaller in size but granting fast data access due to their placement and L3 also called Last Level Cache (LLC). The LLC is shared by every core on the socket. It is not a monolithic structure attached to a bus, but is physically distributed into slices, each attached to a stop on a two-dimensional mesh interconnect. Physical addresses are hashed across slices, so there is no locality relationship between the core issuing an access and the slice holding the data. Any core in the socket can, in principle, evict any other core's data.

The last level cache is the resource most often identified as the dominant source

of interference between colocated processes, since it is shared by every core on the socket and any core can evict any other’s data [7]. Two properties of current server parts qualify how much traffic actually reaches it. L2 is 2 MB per core, considerably larger than the private caches assumed by much of the earlier contention literature. And L3 is non-inclusive: a line resident in L2 need not have a copy in L3, and L3 behaves largely as a backup cache for lines evicted from L2 rather than as a superset of the private caches. A workload whose repeatedly accessed data fits in the private hierarchy therefore generates little L3 traffic and is correspondingly insensitive to what its socket neighbours do to L3.

The cores are not the only accessing cache. On current Intel server parts, data arriving from a network interface is written by the device directly into a portion of the last level cache rather than into main memory using Intel’s Data Direct I/O (DDIO) technology. Also Remote Direct Memory Access (RDMA), known for its high speed, features packet transfers that follow the same path. The intent is to keep the packet path off the memory subsystem and deliver data to the consuming core warm, and at high line rates it is what makes software packet processing viable while using the computational resources not employed for other parallel tasks. The consequence for contention is that the network interface becomes an additional competitor for a resource the tenants are already sharing, with its own occupancy and its own access rate, and the performance of a packet-processing workload becomes coupled to the cache behaviour of processes that never touch the network [7, 8].

Together these mean that a workload whose repeatedly accessed data fits within the private hierarchy may generate almost no L3 traffic, and is correspondingly insensitive to whatever its socket neighbours do to L3. A process may allocate and retain a large dataset while touching only a small subset of it in any interval. The quantity that determines exposure to L3 contention is the active working set per unit time, not the resident set size.

Prior work on network function performance prediction has treated L3 occupancy and cache access rate as the primary predictors of slowdown [7, 9]. Cache occupancy predicts well until available capacity is exhausted, after which access rate becomes the more informative quantity, though a correlation between occupancy and slowdown remains visible [7].

2.1.3 Main Memory

Main memory sits above the last level cache and is built from DRAM modules attached to the socket through its integrated memory controllers. Every core on the socket issues its requests through those same controllers, so memory is shared socket-wide rather than per core, and the network interface joins them as a client whenever a transfer misses the cache. Two properties define its behaviour. Access latency is one to two orders of magnitude higher than a cache hit, so the number of accesses that reach DRAM largely determines a workload’s memory stall time. And the controllers have finite service capacity, so latency is a function of the aggregate request rate across all clients rather than a fixed cost, thus multiple tenants share memory bandwidth.

Memory bandwidth is the rate at which data can be moved between DRAM and the socket, set by the number of memory channels, the transfer rate of the modules, and the width of the interface. It is a property of the socket rather than of any core, and it is not divided among the cores in any fixed way. Every core draws from the same aggregate, so a single core can consume a large share of the available bandwidth if its access pattern allows, and the share any one core obtains depends on the rate of other incoming requests.

Contention for bandwidth differs in kind from contention for cache capacity since it is a queueing issue: requests from all cores arrive at controllers with finite service capacity, and a sufficient aggregate request rate raises the service time experienced by every requester. A workload can therefore be slowed without losing a single cache line.

The independence of this channel from cache contention has been demonstrated directly. Partitioning the last level cache so that a target function holds a private allocation, and then subjecting the machine to bandwidth-intensive competitors, still produces measurable throughput loss in the target while its own cache miss rate remains essentially unchanged [7]. Hence isolating one level of the hierarchy does not isolate the level above it.

Bandwidth contention also removes distinctions that separate cleanly at the levels above it. The memory controller is shared socket-wide rather than per core, so once DRAM is the bottleneck, the placement of a competitor relative to the victim, whether on the same physical core, on a sibling thread, or on a distant core of the same socket, no longer changes the outcome. Signatures that depend on topology under compute-bound contention collapse into a single regime.

2.2 Frequency and Power

Core frequency is not a fixed property of the hardware. It is selected at runtime by platform power management, and the selection depends on how many cores are active, what instruction mix those cores are executing, and the prevailing package power and thermal budget. On Intel parts the achievable turbo frequency is a decreasing function of the number of active cores, and certain instruction classes impose additional frequency limits.

Activity on neighbouring cores changes a target process's clock rate. A core running in an otherwise idle package can sustain a higher frequency than the same core in a busy one. This is a path by which a neighbour affects a workload's latency without any interaction through caches, memory, or the scheduler.

The way frequency scaling interferes with a process can be either enhancing or disruptive. A workload that is idle for much of each interval is held at a low frequency by the governor, because the governor infers low demand from low utilisation. A busy neighbour on the same physical core keeps that core occupied and causes the governor to raise the frequency, which benefits both threads. In this configuration the microarchitectural penalty of SMT sharing and the frequency benefit of a busier core act in opposite directions.

Established models of contention express degradation as a ratio or a regression output that increases with the pressure applied by competitors [10, 11, 7, 9], assuming frequency to be fixed and pinning predetermined governor settings.

2.3 NUMA Nodes

A Non-Uniform Memory Access (NUMA) node is a set of cores together with the memory that is local to them, meaning the memory reachable through their own controllers without traversing an inter-socket link. Access to memory is non-uniform, so the latency and bandwidth a core sees depend on which node holds the page it is accessing. On the platform described in Figure 2.2 each socket forms one node, and the machine presents two.

Access to memory on a remote node traverses the inter-socket link, with higher latency and lower bandwidth than local access, and consumes a link that every core on both sockets shares. A process whose memory is allocated on a remote node is therefore slower in isolation and additionally exposed to contention from a set of neighbours with which it shares no cache.

Allocation policy determines whether this arises. The default first-touch policy places a page on the node of the core that first writes to it, producing local allocation when a process is pinned before it initialises its data and remote allocation when it is not. A competitor placed on the remote socket shares with the target process only the interconnect and the package power budget.

2.4 CPU Time and Scheduling

Not all sources of contention are purely hardware based. Operating systems have agency in deciding how to assign execution time to processes. If more than one runnable thread is assigned to the same logical CPU, the operating system scheduler time-slices between them. The target is descheduled for some fraction of each interval and does not execute at all during that time.

This mechanism differs from every level above it in two respects. It removes cycles rather than degrading the work performed per cycle, so its signature is a reduction in the victim's executed cycles per unit of clock time with the ratio of instructions to cycles left largely intact. And it is the only mechanism in the hierarchy that a purely software-level view of the system can observe, since the scheduler is the entity performing it.

To avoid this issue it is possible to operate CPU pinning, assigning a process to a single logical core and isolating its execution, so that a single process runs without sharing its execution time with anyone else.

2.5 Characterising Contention

Knowing how hardware resources work provides information on the levels at which a server is shared and the mechanisms that produce interference, Table 2.1 collects them.

The mechanisms differ in whether they remove cycles or reduce work per cycle, in whether they act through capacity or through queueing. The sets of neighbours are nested but not identical, so a competitor on the SMT sibling shares every level, one on another core of the same socket shares from L3 onward, and one on the remote socket shares only the interconnect and the power budget. The signatures in the last column are expectations derived from the theoretical mechanisms discussed.

Contention, however, is not determined by the resource alone. Two workloads

Table 2.1: Levels of hardware sharing, the neighbours each is shared with, the mechanism of degradation.

Resource	Shared with	Mechanism	Signature
Pipeline, ROB, ports	SMT sibling	Structural occupancy	Port utilisation, IPC
L1, L2, TLB	SMT sibling	Displacement	Private hit rates
L3, mesh, DDIO	Socket, NIC	Eviction	Miss rate, occupancy
Memory controllers	Socket	Queueing	Miss latency, variance
Frequency	Package	Power budget	Clock rate
Inter-socket link	Both sockets	Remote latency	Remote traffic
CPU time	Logical CPU	Time-slicing	Cycles per unit time

placed identically, with identical neighbours, can experience entirely different degradation, because they interact with resources and other tenants in completely different ways.

A workload’s contention behaviour may be decomposed into two properties. An application’s *contentiousness* is the degradation it causes in its co-runners through its demand on shared resources. Its *sensitivity* is the degradation it suffers when colocated with contentious neighbours. Both are defined as relative IPC loss. Sensitivity is the drop in an application’s own IPC when co-running rather than running alone, averaged over a set of co-runners; contentiousness is the drop it induces in the IPC of those co-runners [11].

The decomposition is most useful when the two quantities are properties of the target application rather than of the pairing of target and aggressor. Measured over co-running pairs, the correlation holds: the ordering of applications by contentiousness is consistent whichever co-runner is present, and the ordering by sensitivity is stable as well. The two orderings do not overlap as their correlation is 0.48, and all four combinations occur in practice, including applications that damage their neighbours while barely suffering themselves [11]. When trying to use a single indicator instead of two the one most often used is last level cache miss rate, it predicts contentiousness with an R^2 of 0.224 against 0.834 for a model built over several resources at once.

Models differ mainly in how they represent the degradation against a function running alone. The Application Slowdown Model starts from the observation that an application’s progress is proportional to the rate at which it reaches the shared cache, which turns slowdown into a ratio of two rates rather than of two execution

times,

$$\text{Slowdown} = \frac{\text{CAR}_{\text{alone}}}{\text{CAR}_{\text{shared}}}, \quad (2.1)$$

where the shared term is measured directly and the alone term is reconstructed by giving the application top priority at the memory controller for short intervals and counting, with an auxiliary tag store, the misses that would have been hits with the cache to itself [10]. MISE builds the same ratio from the memory request service rate and scales it by the fraction α of cycles the application spends stalled on memory, as $\text{Slowdown} = (1 - \alpha) + \alpha R_{\text{alone}}/R_{\text{shared}}$, since only that fraction of execution is governed by the ratio [12].

In SLOMO contentiousness is profiled per application as a vector over the shared levels rather than a scalar, the vectors of several competitors are combined into one aggregate, which for socket-wide quantities is a sum or an average, and the target’s sensitivity, fitted as a function of that aggregate is evaluated as the result [7].

Degradation is often expressed in Instructions Per Cycle (IPC), or in a rate standing in for it, but the correlation between IPC and degradation is mostly assumed. Every one of the discussed models pins each application to its own core, so CPU time, the last row of Table 2.1 and the only mechanism that removes cycles outright, is excluded by construction before measurement begins. When the scheduler can operate freely both placement and frequency scaling can make the system much more difficult to control but core level contention and frequency shifts cannot be completely excluded.

3 Cloud Deployments

Over the years network functions were traditionally implemented on a bare metal device. Functions such as firewalls, a network address translator, a deep packet inspection engine or a gateway arrived as a physical appliance, built around application-specific integrated circuits, installed in a rack and wired into the topology at the point where their service was required. This model carried one important advantage, which is predictability: the throughput printed on the datasheet is the throughput obtained in practical applications.

In this case predictability comes at an economic cost: capacity is fixed at purchase, so an operator sizes each box for its peak load and pays for that peak permanently. Adding a service means adding hardware, with the space, power and cooling that are required. Upgrading means a replacement rather than an update, even when the existing equipment is adequate for every other purpose. Each device comes with its own management interface and its own operational knowledge, and the number of distinct hardware architectures in the network grows with the number of services offered [13].

Network Function virtualisation was proposed in order to grant versatility in infrastructures. Implementing the function in software, running it on the same commodity servers used everywhere else in the data center, and letting the network topology be a matter of configuration rather than of cabling proves to be much more cost efficient. Figure 3.1 shows the reference architecture ETSI defined for this, with the functions themselves running on a virtualised infrastructure and a separate management and orchestration stack deciding what runs where [14].

The advent of softwarized solutions grants much more flexibility as software evolves independently of the hardware it runs on, so a server's life cycles last longer since they are able to run processes over time without needing hardware upgrades. Functions can be placed anywhere there is capacity, and moved, so the same physical machine can run different processes at any given time and provisioning becomes elastic, so capacity follows demand instead of being sized once for the worst case. Software based solutions solved scalability issues while optimizing the existing hardware

3 *Cloud Deployments*

usage. Infrastructures are much more scalable since commodity servers are produced in volume, their components are interchangeable and priced competitively, and using them reduces the number of different hardware architectures an operator has to keep available. Using bare metal, providers tended to overprovision, and the allocated resources sat mostly idle in exchange for the headroom they provide against traffic spikes and component failures. Because several services draw from a common pool and their failure modes are independent, that headroom can be shared instead of duplicated, and the spare capacity can be reclaimed for other work: consolidating the daytime load onto fewer machines overnight, and turning the rest off or renting them out [14]. The management layer that makes this feasible is called Cloud Computing, that is, the software that serves resources from a pool of machines on request and takes them back when they are no longer needed. High adaptability comes at the cost of a more unpredictable behaviour as a function that no longer owns its hardware has a capacity determined by what resources are dedicated to its execution at a given instant. Measurements of end-to-end performance on Amazon EC2 found TCP and UDP throughput fluctuating between zero and 1 Gb/s at a granularity of tens of milliseconds confirming highly erratic behaviour. Instances may reach delay variations up to a hundred times larger than the propagation delays involved, on a network that is not heavily loaded [15]. Cloud computing also allows for renting machines and computing power from providers to users. Often in a public deployment the party that owns the server and the party that owns the function are different. The server owner allocates resources; the tenant deploys functions into them and supplies the traffic. The owner usually cannot inspect processes and functions deployed on their proprietary hardware. High-speed functions drive their network interfaces in kernel-bypass mode, which removes the traffic from the operating system's view and therefore from the server owner's; specific applications exist to keep traffic and function internals unreadable to the infrastructure provider [4]. An operator who wants to know details about utilisation of its machines has to find out from the outside, using what the platform exposes about the hardware rather than data provided by the function itself.

3.1 Virtualisation

Network functions and cloud services are deployed under several execution models. A function may run as a bare-metal process, inside a container, or inside a virtual machine. These differ in the isolation boundary they establish and in the amount

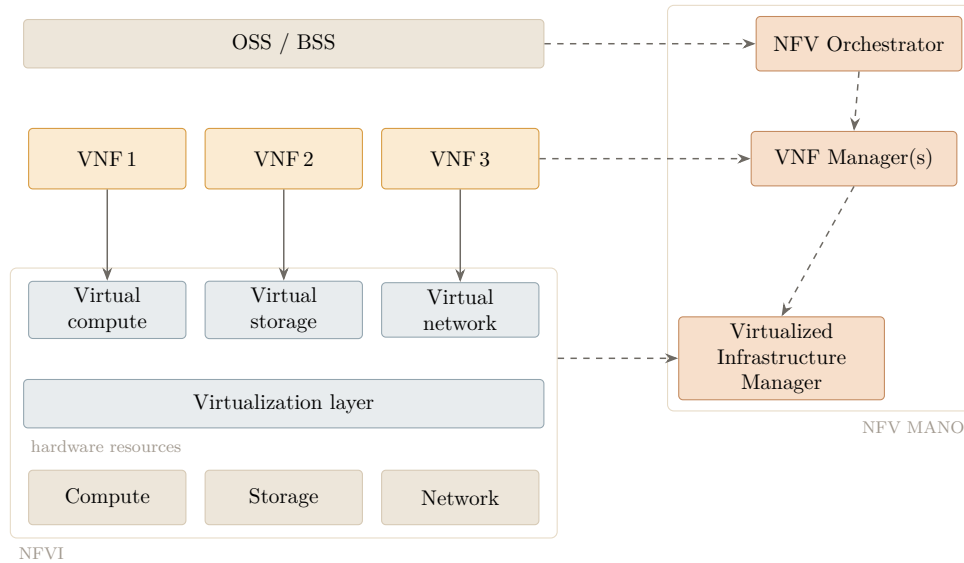


Figure 3.1: NFV Framework.

of software between the workload and the hardware, but they do not differ in what the hardware shares. A container and a virtual machine pinned to the same logical CPU are subject to identical microarchitectural interference, because the sharing described in the previous chapter occurs below the level at which either abstraction operates.

Virtualisation is the practice of presenting a resource in a virtualised form that differs from its physical one, through a layer that translates between hardware and software and enforces the separation between the parties using it. The layer intercepts the operations a workload performs on the lower levels and maps them onto the physical resources. Each workload behaves as though it were a physical machine although it is provided a mapping from its application layer to the hardware layer.

The two main implementations of virtualised instances are Virtual Machines (VM) and Containers.

Virtual Machines Here the abstraction is a whole computer. The guest using the machine receives virtual CPUs, a virtual physical memory, and virtual devices, and runs its own operating system kernel on top of them. The layer managing the connection is called hypervisor, or virtual machine monitor. It is conventionally divided into type 1, which runs directly on the hardware, and type 2, which runs as an application inside a host operating system.

The mechanism behind the hypervisor is interception. Ordinary guest instructions

3 Cloud Deployments

execute directly on the physical processor at full speed. Operations that would affect the state of the machine as a whole, such as programming a device or changing the page tables, must not be allowed to take effect as written, so they are trapped and carried out by the hypervisor on the guest's behalf. Current processors tackle this issue by providing a separate guest execution mode with hardware support for the trap boundary, so the transition between guest and hypervisor is an architectural event rather than emulation.

The isolation obtained this way is strong. The guest has its own kernel, so a compromise or a crash inside it does not directly reach the host or the other guests, and the interface the guest is allowed to interact with is restricted. Virtual machines grant very good isolation but are heavyweight as every user carries a full operating system in memory and leverages a second layer of address translation on every memory access it makes.

Containers Here the abstraction shifts from whole machines to a restricted view of the already running machine. A container is a set of ordinary processes on the host kernel, characterised by two main features:

- Linux namespaces: providing the container a private set of system resources such as an owned process identifier space, its own mount table, its own network interfaces, routing table and port numbers and its own user identifiers and hostname;
- Control groups: describing how much of each resource the container may consume and how much is allocated at any given time.

Differently from virtual machines system calls from the container go straight into it with no intermediary, and there is no second address translation.

Figure 3.2 shows the difference between the three implementations. Not having a guest kernel makes this implementation lightweight. Containers start in milliseconds, their memory footprint is the footprint of the application they run, and their execution overhead is negligible compared to the bare-metal counterpart. The isolation is correspondingly weaker: every container on a machine shares a single kernel, so a kernel defect is a defect for each of them, and the interface exposed to a container is the entire system call surface rather than a handful of virtual devices. In current practice this trade-off is usually resolved in favour of containers as they are the dominant deployment model.

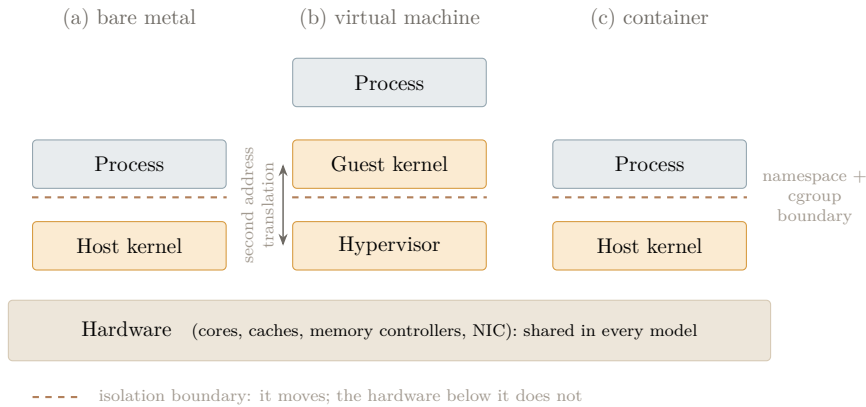


Figure 3.2: Different execution models, from bare metal to virtualisation

3.2 Control Groups

A control group (cgroup) is a Linux kernel facility that collects processes into a named group and accounts for, limits and prioritises the resources that group consumes as a whole. Groups form a tree, processes inherit the group of their parent, and the limits attached to a group apply to everything inside it including any child process.

Control groups are meant to tackle resource sharing issues inside the kernel. The kernel’s natural unit of fairness is the thread; when the machine is shared between parties the thread proves to be an unfair and inefficient primitive. A scheduler that divides CPU time evenly between runnable threads rewards a tenant for starting more of them, and the same holds for memory and I/O. Control groups render resource sharing programmable rather than allocating more for a tenant that uses many threads at any given moment. Container runtimes are built directly on this: a container is a set of namespaces defining how processes interact, plus a cgroup defining what hardware they can leverage.

When deploying a container three controllers govern how many resources will be allocated for the specific instance:

- **cpu**, which governs how much CPU time the group receives. `cpu.weight` sets a relative share, applied only when there is competition for the CPU: a group with twice the weight of another gets twice the time when both are runnable, and gets everything it asks for when the machine is otherwise idle. `cpu.max` sets an absolute ceiling, expressed as a quota of microseconds per period, and is enforced whether or not the machine is otherwise idle.

- **cpuset**, which governs placement. `cpuset.cpus` restricts the processes in the group to a set of logical CPUs, and `cpuset.mems` restricts their memory allocations to a set of NUMA nodes.
- **memory**, which caps the resident capacity the group may hold. It bounds how much memory is occupied, not how fast it is accessed.

3.3 Orchestration

A single server can be managed by hand. Multiple servers require some orchestration tool, and since the properties that make cloud deployment attractive are elasticity in deploying workloads wherever capacity exists and moving them when demand changes, a cluster of servers is always required. Orchestration is the software layer that manages that cluster: it manages workload allocation, creates and removes resources as load varies, and connects the machines to each other and to the network. The operator states what should be running and under what constraints; the orchestrator is responsible for making the cluster match the desired state at all times. In the ETSI framing of Figure 3.1 this role is split between the orchestrator and the virtualised infrastructure manager, the first concerned with services and the second with the compute, storage and network resources available [13].

Kubernetes is the system that has become the default vehicle for this in general purpose infrastructure.

Figure 3.3 shows a Kubernetes cluster: featuring a control plane and a set of worker nodes, each node being a machine that runs workloads. The unit of deployment is not a container but a *pod*: one or more containers that share a network namespace and a lifetime, and that are always placed together on one node. The component that chooses a node for each pending pod is the *scheduler*. The agent that runs on every node, receives the pods assigned to it, instructs the container runtime to start them and applies their resource settings, is the *kubelet*.

Resources enter this model through two numbers defined per container. A *request* is the amount the container is assumed to need, and it is what the scheduler uses when deciding whether a node has room: a node is eligible if the sum of the requests already placed on it plus this one fits within its capacity.

limit is a ceiling on what the container may actually consume. CPU is expressed in fractional cores, with 1000m meaning one CPU, and it denotes an amount of CPU time per unit of wall-clock time rather than a particular core. The kubelet

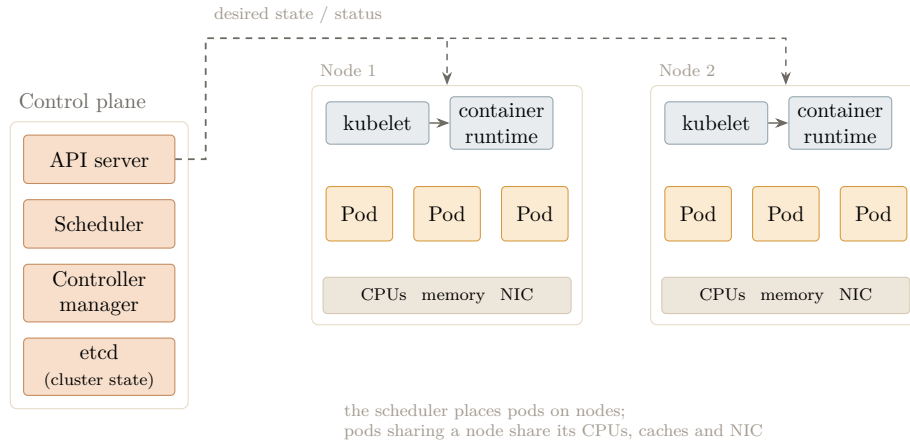


Figure 3.3: Kubernetes general architecture

translates the request into `cpu.weight` and the limit into `cpu.max`. The relationship between the two numbers determines a pod’s quality-of-service class: equal requests and limits for both CPU and memory across all its containers make it *Guaranteed*, a request without a matching limit makes it *Burstable*, and declaring neither makes it *BestEffort*. Guaranteed pods are the last to be evicted when a node is shut down.

None of this, under the default configuration, results in a workload owning a core. The kubelet’s default CPU Manager policy places every pod on the node into a single shared pool of logical CPUs. The scheduler places the pod by comparing aggregate requested CPU against aggregate capacity, without reference to the node’s topology, and the kernel then time-slices among processes that end up co-resident. A CPU request is a claim on an amount of CPU time, not an allocation of a particular core.

Dedicated cores require opting in to the *static* CPU Manager policy. A container under static management receives an exclusive cpuset, and the shared pool that serves everything else on the node shrinks by that many CPUs. The policy is set at node level and changing it requires restarting the kubelet, so it is a property of how the cluster was built rather than something a tenant can request per workload.

3.4 Cloud to Hardware Interaction

Underneath management layers lies the interconnection between high level user instructions and the hardware that executes them. From software to low level instructions go through address translation, dedicated interrupts and interaction with peripheral devices.

3.4.1 Address Translation

An ordinary process addresses memory virtually. The memory management unit reads a tree of page tables to convert each virtual address into a physical one, and caches the results in the translation lookaside buffer, so that the access is performed only on a miss. On current x86 the tree has four levels, so a miss costs up to four memory references before the access itself can proceed.

A guest inside a virtual machine cannot be allowed to control this tree, because having no knowledge of physical addresses renders it impossible for the guest to carry out the translation. The hardware therefore provides a second, nested translation: the guest's page tables map the guest virtual addresses to guest physical addresses, and a second set of tables owned by the hypervisor, maps guest physical addresses to host physical ones. Both are read by the hardware without hypervisor involvement, which is what makes the scheme fast enough to use, but the access becomes two-dimensional: every level of the guest access produces a guest physical address that must itself be translated through the second table. The worst case grows from four references to twenty-four [16].

With virtualisation the translation structures themselves occupy cache, so a virtual machine's cache footprint includes page table levels that a bare-metal process would not have, and its sensitivity to a neighbour evicting them is correspondingly higher. And pressure on the TLB becomes more expensive, because each miss costs more. This is the main reason large page mappings are close to mandatory in packet processing deployments: a 2 MB page removes a level from both the guest and host access and covers roughly five hundred times the address range per TLB entry.

3.4.2 Interception

When a guest performs an operation the hypervisor must handle, such as touching an emulated device register or receiving an interrupt that the host owns, the processor leaves guest mode and enters the hypervisor. This transition, a VM exit, saves the guest's architectural state, loads the host's, runs the handler, and reverses the process on the way back.

The direct cost amounts to a few hundred to a few thousand cycles depending on the exit reason. The transition not only has an intrinsic cost but also drains the pipeline and displaces state in the caches, the TLB and the branch predictors, so the guest resumes on a colder machine than the one it left. For workloads that depend heavily on private caches, an exit is a small eviction event rather than a

delay. Exits also distort measurement, since the cycles and instructions attributable to the transition belong to neither party cleanly.

Some workarounds ensure that these exit events are rare. Polling the network interface instead of taking interrupts removes the most frequent source. Assigning the device to the guest removes the emulation exits entirely. Disabling processor idle states removes the wake-up latency of a core that has been allowed to sleep, and isolating cores from the host's scheduler ticks and read-copy-updates removes the host's own periodic work from the cores the guests are using [7].

3.4.3 The I/O Path

In order to make packet flow possible, virtualised implementations must provide a path to peripheral devices. Many different solutions are available and each differs in cache access method and per packet CPU cost as shown in Figure 3.4.

- **Emulation:** the hypervisor presents a model of a real network card, and every access the guest driver makes to a device register traps. The guest needs no modification, but the per-packet cost is several exits, which perform poorly at high throughput rates.
- **Paravirtualised devices:** the guest driver is entirely virtual and communicates through shared-memory ring buffers, so the register accesses disappear and only a notification remains. A host-side component, typically a `vhost` kernel thread or a userspace equivalent, takes packets off the ring and puts them on the actual wired path. This solution requires the presence of a host thread working simultaneously on the tenant's behalf and outside the tenant's cgroup, thus being a competitor for cache and bandwidth like any other, with a load scaling with the tenant's packet rate.
- **Software switching:** when different guests connect between each other and with the outside under a policy, a virtual switch such as Open vSwitch or VPP runs on the host and forwards between them. This is a full packet processing workload featuring dedicated flow tables, cache footprint and cores, colocated with the functions it serves.
- **Device assignment and SR-IOV:** a network interface that supports single-root I/O virtualisation presents itself as one physical function plus a number of virtual functions, each appearing as an independent PCIe device with its own

3 Cloud Deployments

queues. A virtual function can be mapped directly into a guest, which then drives it as though it owned the card. Safety comes from the I/O memory management unit, which translates the addresses the device uses for direct memory access, so the device can write into guest memory and nowhere else without the hypervisor mediating each transfer [17]. The hypervisor is then much lighter on the data path, and the host spends fewer cycles per packet. The cost is residual latency on top of the bare metal implementation, measured at roughly one microsecond on tail latencies [18], and a loss of flexibility, since the number of virtual functions is fixed by the hardware and a guest bound to a physical device is harder to migrate.

Containers face the same set with different names. The default is a virtual Ethernet pair into a host bridge, where the host kernel handles every packet; the alternative is to hand the container a virtual function through the orchestrator’s device plugin, or to give it a shared-memory port into a userspace framework such as OpenNetVM or FastClick, in which case the container’s packets never enter the host network stack at all.

On current server parts, data arriving from a network interface is written by the device directly into the last level cache. With an assigned virtual function, the device writes the packet and the guest core reads it, and no other party is involved. With a paravirtualised path or a virtual switch, a host core reads and writes every packet first, so the same traffic occupies the cache twice over and adds a second consumer of memory bandwidth. The I/O model therefore does not just set a per-packet overhead, it sets how many clients the shared memory subsystem has.

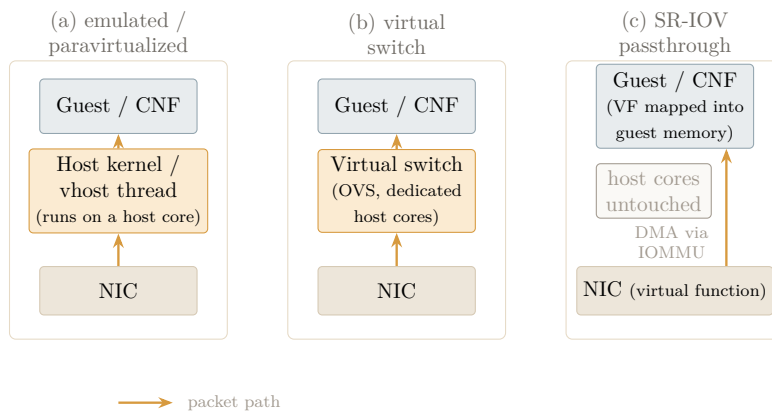


Figure 3.4: Different I/O solutions in virtualised deployments

3.5 Telemetry in virtualised Environments

The cloud stack may impose several challenges to observing processes. Every added layer distances the software from the hardware and in real cases user and provider are both limited in inspecting the machines that run a virtualised function. A way to understand how the software processes are impacting the machine itself for a provider would be to obtain data on hardware execution. To do so a provider may ensure that user issued functions run in specific ways: for example pinning them to a core shows the execution pattern in its clearest form. It is the control applied most as it addresses CPU time and nothing beneath it, and its effect on everything above it is to make interference harder to detect.

Time-slicing is interference the operating system performs itself, so the operating system reports it. Once a container holds a core alone, what remains is a reduction in the work performed per cycle. A pinned container running continuously, when observed from the orchestrator, reports full utilisation of its core whether it is retiring four instructions per cycle or one. For packet processing functions the situation is even more unclear, since these busy-poll their interfaces and hold a core at close to one hundred percent reported utilisation at every offered load, including no load at all [4].

The stack also multiplies the parties drawing on the hardware, and it does so with parties the tenant never declared. A paravirtualised data path puts a host thread on a host core doing per-packet work whose rate is set by the tenant's traffic. A virtual switch is a second packet processing workload colocated with the first. The network interface writes into the last level cache on its own account. None of these appear in any tenant's accounting, and their combined footprint is part of the slowdown experienced by any individual tenant.

Underneath that, the units a user can name and the units the hardware arbitrates do not coincide. The stack names pods, containers and control groups. The hardware arbitrates over a socket's cache, a socket's memory controllers, a physical core's execution ports and a package's power budget.

Finally, the layers move the baseline as well as the signal. A function measured on bare metal, in a container and in a virtual machine yields different results with no competitor present at all, because nested translation, exits and the chosen I/O path each add work of their own. Whatever an operator learns to recognise as the signature of a neighbour has to be recognised against a reference established for that deployment, not for that function.

3 Cloud Deployments

The processor still counts cycles, instructions, cache accesses and stalls regardless of how many abstractions have been stacked above it, and it counts them for what is actually executing rather than for what the orchestrator deems to be executing. Leveraging this information would not be affected by the limits of cloud orchestration.

4 Observing Performance Under Contention

A function deployed on a shared server processes fewer packets than the same function on an empty one. In order to understand how much was lost, and which shared resource caused the loss, infrastructures must be equipped with telemetry tools. Considering the packet rates modern software data planes sustain, observability solutions are evolving around costs and accuracy tradeoffs.

Many technical solutions may also prove not to be practical as they need the source code of the observed function, access to insert a probe, or the cooperation of whoever is running the process. In a public deployment the boundary between a tenant and the provider is clear cut, so tools that require access to the other party's domain might be unusable.

A telemetry application must therefore take into account three fundamental constraints: cost, accuracy, and feasibility.

4.1 Network Telemetry

The direct way to know how a network function is performing is to look at its packets. By counting them and timing them the throughput and latency are observed without any model in between.

The earliest form is a counter on the device. A network interface counts packets and bytes received, transmitted and dropped, and a management protocol polls them every few seconds. This solution is lightweight and reports volume of traffic at a point in time but doesn't give deeper information on the network.

Further improvements were achieved with the emergence of Software Defined Networking (SDN) and Programmable Data Plane (PDP). More open control plane and data plane programming capabilities have reshaped the traditional network measurement with SDN reconstructing the network control plane and data plane without

4 *Observing Performance Under Contention*

changing the foundation of the IP network. On one hand, these techniques improve the flexibility of the network measurement architecture. On the other hand, the flow collection and sampling methods used in traditional network measurement techniques have been preserved. Compared to traditional network measurement, software-defined measurement has great advantages in openness, transparency, and programmability.

Network telemetry has been widely considered as means to gain sufficient network visibility with better overall performance than traditional network measurement technologies. In-band network telemetry is one of the main representatives of network telemetry: different from the traditional network measurement and software-defined measurement as it combines data packet forwarding with network measurement. In-band network telemetry collects the network status by inserting metadata into packets: namely business packets, adding information to the headers, to carry telemetry instructions or telemetry data, and the source node embeds tags or instructions in the business packets to indicate the network information that needs to be measured. The data plane fills the telemetry information into the business packet while matching and forwarding and on the last hop the packet reports all telemetry data to the server. This approach changes the network measurement process from control plane driven to data-plane driven and reports information per packet [19].

Inside a server the same idea attaches to the software path rather than to the wire. Various approaches tackle the issue by inserting probes in different sections of the network.

NFVPerf uses port mirroring on every physical machine to capture all incoming and outgoing packets (including packets between VMs on the same host) in the NFV deployment. The mirrored packets are then analysed by a script that performs deep packet inspection to compute various metrics of interest [20]. Mirroring packets requires using memory copies, so NFVPerf makes intense use of memory on the machines where it is deployed.

PPTMon measures the packet processing time of a packet as the difference of timestamps between ingress and egress when the packet passes through a VNF, then stacks the measured packet processing time in the packet data [21]. Every packet is touched twice and every crossing generates an event, so the method might scale poorly in a saturated data plane.

NFV-VIPP collects metrics from inside functions built on kernel-bypass frameworks, whose exclusive resource management hides the packet path from ordinary system statistics. By exposing the internal stages of the processing loop it recov-

ers information no external observer has, and it shows that the cycles spent in the receive-polling stage can be used to estimate the load on the node [22]. Such implementation comes at the cost of integrating the tool with each function, and each collection thread occupies a core of its own [23].

Figure 4.1 shows the different telemetry solutions compared.

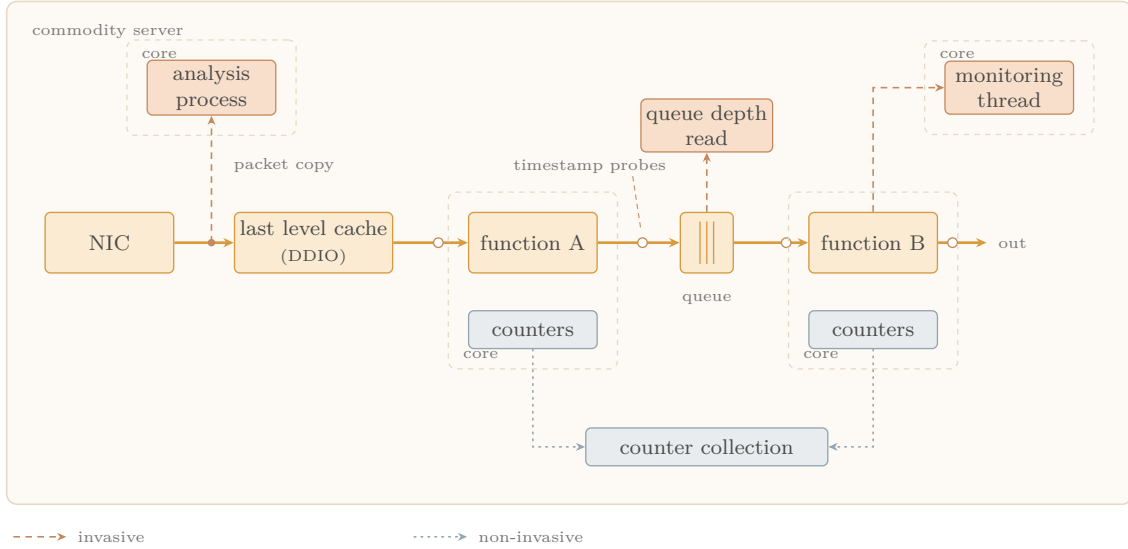


Figure 4.1: Various telemetry solutions.

4.2 The Observer Effect

Observing a process may result in poor-quality data if the data collection alters the performance of the system being measured; the scientific literature calls this the observer effect [24].

Altering the performance may occur in different ways such as stealing cycles, filling queues or evicting useful data from memory. An application that runs on the same machine competes for the same cores, the same cache and the same memory bandwidth as the function it watches. When the application does work once per packet, the workload scales with the traffic, and it increases in critical scenarios.

A 10 Gbps link carrying minimum-size packets delivers 14.88 million packets per second, one every 67 nanoseconds [1]. As network services response time is increasingly getting lower in the order of sub-milliseconds, direct performance measurement can heavily stress the software data plane, resulting in packet processing time degradation and SLA violation such as low throughput and high latency [23].

The telemetry tool also consumes cache and memory bandwidth, which are the resources through which colocated workloads interfere with each other. Per-packet collection therefore does not only reduce the performance being measured. It adds a competitor to the scenario under study, and the observation and the phenomenon become more entangled.

To circumvent these obstacles, a line of work takes a novel approach by exploring low-level features to derive insights into service performance. In particular, some approaches infer the performance of the software data plane from cache features and a wider set of low-level features, selecting relevant features by investigating system-level contention, and employ machine learning for throughput prediction [23]. The issue then moves from per packet analysis to inspecting batches of packets and system behaviour over time mitigating the observer effect.

4.3 Inferring Performance from Hardware Events

The processor counts events whether or not anyone reads them. Cycles, instructions retired, cache accesses at each level, translation lookups, branch outcomes and stall reasons are accumulated by dedicated registers as a byproduct of execution. A function supplied by a third party, driving its interface in kernel-bypass mode and holding its packets in memory the provider cannot inspect, still leaves a complete trace in these registers.

This turns a measurement problem into an inference problem: given a vector of event rates observed over an interval, estimate a quantity that was never measured directly.

The target of this inference is network performance indicators, such as throughput and latency, while also providing information about resource usage. The execution profile of a packet processing function changes in a characteristic way when incoming traffic varies, producing different effects on the underlying hardware.

A predicted curve would refer to the function behaviour rather than to the traffic, so a reading taken at an unknown load can be located on a specific one. Building a representative feature vector for each known condition and comparing new observations against those vectors by similarity reveals patterns in values of the input rate, the packet size distribution and the arrival pattern without directly accessing packets. This approach has several advantages. Regardless of the peculiarities of network and system configurations, these hardware information are always available and can be readily collected via system calls and encryption or other forms of secu-

ity do not hinder the collection of these low-level features. Compared to in-band measurement, collecting low-level features only happens at the hardware registers and for this reason is much lighter on the data plane. Many low-level features are already collected in the system registers, the additional cost is only in reading them [23].

While providing information on network performance, hardware events are highly informative on the resource sharing between different colocated functions. There are various ideas on how to leverage these data to infer the workload sharing among competing processes.

One approach reduces the competition to a single number, such as the rate at which neighbours access the cache or the size of their working set, and expresses the target's throughput as a function of it [9]. The approach is simple: one scalar can be profiled once per application, and the pressure of several neighbours can be composed by addition. However the memory system is not a single resource: a scalar chosen to track cache capacity doesn't carry information on bandwidth queueing just as tracking cache access rate overlooks cache occupancy.

Instead of using a single scalar one could use a feature vector and separate the memory subsystem into distinct points where contention arises and gives each its own features [7]. This approach aims at identifying a core set of hardware events that are especially relevant to contention and leverage them to profile a function. Functions might be tested against several competitors in order to better understand their behaviour when exposed to different kinds of stress and then predict performance in a small range of similar cases.

A much harder task is generalising from one function to a chain of them. Using models like augmented neural networks chosen for their tolerance of non-linear relationships between many inputs, it is possible to look for global correlations between the network behaviour and hardware events that go beyond the specific function.

4.4 Performance Monitoring Units

Hardware counters are what turns a measurement problem into an inference problem, and they are exposed by the Performance Monitoring Unit (PMU), a block of the core whose only task is to count what the rest of the core does.

The part of the unit leveraged is its set of Performance Monitoring Counters (PMC). Modern units carry other facilities alongside them: last branch records, which retain a short history of taken branches; precise event based sampling, which

4 *Observing Performance Under Contention*

captures architectural state at the instruction responsible for an event rather than at the interrupt that follows it; and processor trace, which reconstructs the executed control flow. These answer questions about where inside a program something happens, at a cost proportional to how often it happens, so they belong to program analysis rather than to continuous telemetry [25].

Counters come in two kinds. Fixed-function counters always count the same event and need no configuration. Intel parts provide three of them per logical processor: core clock cycles, reference clock cycles, and instructions retired. Programmable counters are configured by whoever reads them, and each can be pointed at one of several hundred defined events covering the instruction pipeline, every level of the cache hierarchy, the translation lookaside buffers, branch prediction and dispatch to individual execution ports.

Physically the counters are model specific registers. A programmable counter is paired with an event select register holding the event code, a unit mask that selects a variant of the event, bits restricting the count to user or supervisor mode, and an enable bit [26]. Access to these registers through `RDMSR` and `WRMSR` is privileged, so configuration happens in the kernel, while the current value of a counter can also be read from user space with `RDPMC` when the operating system allows it.

Neither the number of counters nor their width is fixed across the product line, so both are queried rather than assumed. The `CPUID` leaf reserved for architectural performance monitoring reports the version of the unit, the number of programmable counters available per logical processor, their width in bits, and the same two quantities for the fixed counters. On current parts the counters are 48 bits wide, which is enough that a counter rarely wraps within a collection interval. When one does overflow it raises an interrupt, and the software reading it has to record the wrap for the count to remain meaningful [25].

The events themselves can either be already defined or customized to the specific processor. A small set of architectural events keeps the same encoding and the same meaning across generations: unhalted core cycles, instructions retired, unhalted reference cycles, references and misses at the last level cache, and branch instructions and mispredictions retired. Everything else is defined per microarchitecture and listed per model in the vendor manual [26]. The model specific portion is much more detailed, including per-level cache behaviour, port utilisation and requests leaving the core, and it has to be remapped when the fleet is not uniform.

Counters can be read in two modes. In counting mode a counter accumulates over an interval and is read at the end of it, producing one aggregate per interval.

In sampling mode an interrupt is raised every N events and the instruction pointer is recorded, producing an attribution of events to locations in the code. Counting describes how a workload behaves over time and is the mode fit for continuous telemetry.

Reading a counter through the standard kernel interface issues a system call, on the order of a few thousand cycles. The cost is charged per read rather than per event, so at an interval of tens or hundreds of milliseconds the cycles spent collecting are negligible against the cycles a packet processing function spends on traffic. This is the property that separates counter collection from in-band collection, whose cost follows the packet rate.

The number of programmable counters is eight per physical core, each logical processor seeing four when both hardware threads are active. When more events are requested than there are counters, the kernel time-multiplexes them, scheduling each event onto a counter for part of the interval and scaling the partial count to estimate a full-interval value. The estimate carries an error that grows with the ratio of requested events to available counters and with the variability of the workload. A counter set intended for continuous collection therefore has to fit within the hardware, and a larger set has to be split across separate runs of the same condition.

4.4.1 Counter Affinity

A counter reading is defined by three features and each one defines what information the counter carries.

Scope The set of executions the counter accumulates over. The event is the same in every case while the entity observed changes.

- *Thread or process scope* accumulates only while that task is on the processor. The number describes the target's own progress. It requires knowing the task identifier.
- *Control group scope* accumulates while any task in the group is on the processor. The number describes one tenant in aggregate, which is the natural unit for a container.
- *Core scope* accumulates everything that executes on a logical processor, including tasks belonging to other tenants. The number describes the core, not

4 Observing Performance Under Contention

any one workload.

- *Socket scope* accumulates in structures outside the cores and cannot be attributed to a task at all.

The event select register carries a further distinction inside these scopes, since the user and supervisor bits determine whether work performed in the kernel on the task's behalf is counted along with the task itself.

Location

- *Core counters* measure a shared resource as experienced by one consumer: cycles spent stalled on memory, hits and replacements in the private caches, misses that reach the last level.
- *Uncore counters* measure the shared resource itself: traffic through the coherence agents attached to the cache slices, read and write bandwidth at the memory controllers, occupancy of the interconnect.

A core counter shows that a workload is stalled waiting on memory, but a workload that has become more memory intensive on its own produces the same reading as one whose neighbours have saturated the controllers. An uncore counter shows that the controllers are saturated, but attributes the pressure to a socket rather than to a tenant. The quantity an operator would want, one tenant's share of a contested resource, is not exposed by either. Uncore counters are also harder to obtain: they live in a separate set of registers with their own access mechanism, they require higher privilege, and they are frequently unavailable in virtualized and containerised environments. Where they cannot be read, bandwidth contention can be characterised only from the target's side.

Normalisation

- *Per unit of time*. Sensitive to how much service the workload received during the interval. Appropriate for detecting cycles removed by the scheduler, and for comparing against an external rate such as offered load.
- *Per instruction, or per thousand instructions*. Removes the amount of work from the measurement and leaves the efficiency with which it was performed. Appropriate for detecting degradation that acts per cycle rather than by removing cycles, such as displacement in a private cache.

- *Per cycle.* Instructions per cycle is the standard case, with contention for the pipeline appearing while both cycle and instruction counts stay unremarkable if considered on their own.

The choice between the first two is also a choice between the two fixed clock counters, since core cycles follow the frequency the core is actually running at while reference cycles advance at a fixed rate. A rate computed against the first is blind to frequency scaling and a rate computed against the second is not.

Individual counters are numerous and hard to interpret in isolation. A hierarchical decomposition of pipeline behaviour offers a more compact view [3]. Every cycle the processor has a fixed number of issue slots available, and each slot ends in one of four states: it retires a useful micro-operation, it is wasted on a mispredicted path, it is empty because the front end failed to deliver work, or it is blocked because the back end could not accept work.

The value of the decomposition here is that its categories correspond to distinct sharing mechanisms. Competition for execution ports and reorder buffer entries appears as core-bound stalls, competition in the cache hierarchy and at the memory controllers as memory-bound stalls, and competition for branch prediction state as bad speculation. That correspondence makes it a candidate for distinguishing which mechanism is acting, rather than only confirming that something is.

4.4.2 Counters Under Virtualization

The virtualized layer does not change what the hardware shares, but it does change access to resources and which entities can perform such accesses.

Containers share the host kernel. A container is a set of ordinary processes with a restricted view of the system, so counters for a containerised workload are collected exactly as for any other process, subject only to permission. There is no separate monitoring unit and no additional layer of attribution.

Virtual machines interpose a hypervisor. Whether a guest can read counters at all depends on whether the hypervisor exposes a virtual monitoring unit, which is a configuration choice rather than a guarantee. Attribution is also less clean, since the cycles spent entering and leaving the hypervisor belong to neither the guest nor the host in an obvious way.

For an infrastructure operator this asymmetry is less restrictive than it appears, because the operator reads counters from the host and does not need access from inside the guest.

5 Setup and Methodology

Owning the machines themselves grants the ability to inspect their low level performance, since the hardware accumulates cycles, instructions, cache accesses and stall reasons as a byproduct of execution, the accumulation is readable by the owner, and reading it requires no entry into the tenant. A function supplied by a third party, driving its interface in a manner the provider cannot inspect, leaves a complete trace in registers that belong to the provider. A server with multiple processes running at the same time features a set of neighbours and each of them produces interference by sharing resources; this interference then influences performance, which correlates competition for resources with high level performance in networks. For each level of sharing, counters move once a neighbour is deployed in colocation highlighting a distinct fingerprint: private hit rates fall under a sibling thread, miss rates rise under a competitor elsewhere on the socket, and executed cycles per unit of clock time fall once the scheduler begins to time-slice. The magnitude of the impact this sharing has on high level behaviour is not captured although it is the unit in which a service level agreement is defined, so a function whose obligation is expressed in milliseconds of tail latency benefits from mapping low level features to high level performance indicators.

The deterministic route to the missing quantity is analytic as service time can be written as the number of instructions a unit of work requires, multiplied by the average number of cycles each instruction costs and divided by the rate at which the core advances,

$$T \approx \frac{\text{instructions} \times \text{CPI}}{f}, \quad \text{CPI} \approx \text{CPI}_{\text{base}} + \sum_i m_i p_i,$$

with the cycles-per-instruction term decomposed into a base cost and one contribution for each level of the hierarchy, each contribution being the product of a miss rate m_i and a penalty p_i . Every quantity on the right-hand side of the function is measurable in principle, but is highly dependent on the workload under analysis.

Neither the penalties nor the miss rates are constants of the function. The penalty

5 Setup and Methodology

is the portion of an access latency the pipeline cannot optimise, which varies with how many misses are outstanding and how much of their cost the core overlaps with useful work, while the miss rates are coupled to one another, since a neighbour evicting lines from the last level cache raises the miss rate there and lengthens the service time of every miss that reaches the controllers. Each term is thus a property of the workload and neighbour pairing rather than of the workload itself, and evaluating the expression under a given placement requires measurements taken under that placement.

The highest variable quantity at any given time is frequency, since it is a quantity selected at runtime from the activity of the entire package, which makes it a function of the neighbour rather than a constant fixed before the neighbour arrives. A busy sibling thread raises the clock of the core it shares, so that the microarchitectural penalty it imposes and the frequency it grants act against one another and the net effect on latency cannot be predetermined. Established models of contention avoid the difficulty by holding frequency fixed and pinning the governor before measurement [7, 9, 11], which removes from the account the one channel capable of reversing the direction of the result.

For these reasons the expression is not a means of obtaining the quantity so much as a statement of what would have to be known in order to obtain it, and each of its terms becomes knowable only once the neighbour is known.

What the operator holds, then, is a signal that is cheap to collect, present on every host, obtainable without cooperation from the workload and indifferent to what the workload is, while what the operator lacks is the quantity written into the service level agreement, which is the tenant’s latency.

In order to observe general patterns that are common across many workload-neighbours pair an experimental campaign has been conducted. In these experiments placement is the independent variable, both because placement is the decision an orchestrator actually makes and because it logically follows the way Table 2.1 presents hardware sharing based on topology, and the variable is carried across network functions of increasing realism, from a synthetic client driving a proxy chain on a single host through to external traffic arriving over the network interface. Counters and the key performance indicator are recorded second by second against a common clock, so that the campaign is representative of the evolution of a service over time, with the final goal of deducing the correlation between high and low level performance in different scenarios.

Figure 5.1 offers a single run as evidence that the relationship exists: a HAProxy

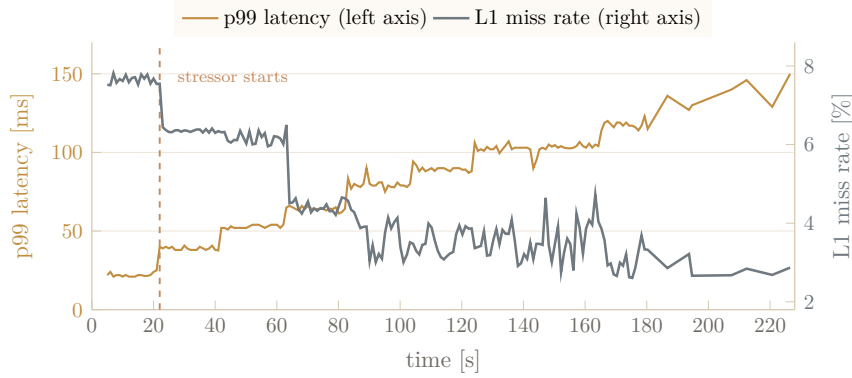


Figure 5.1: P99 latency and the L1 cache miss rate are plotted against the escalating stress phases of the run.

run under a memory-bound stressor placed on the target’s own logical CPU, in which the p99 rises from 22 ms to roughly 150 ms across the escalating stress phases while the first-level miss rate tracks that same escalation with the opposite sign, at a Pearson correlation of -0.92 . The magnitude of the correlation indicates that the counters carry the information, while its sign, which is not the one the mechanism on its own would suggest, is the reason the mapping is measured rather than derived from the terms of the expression above.

5.1 Contention Experiment Topology

All campaigns run on a single dual-socket server built around the Intel Xeon Gold 6442Y, a Sapphire Rapids part carrying 24 physical cores per socket with two-way simultaneous multithreading, so that the operating system sees 96 logical CPUs. Each core holds a private 48 KiB first-level data cache and a private 2 MB second-level cache, while the 60 MB last level cache is shared across the cores of a socket, and cores range from 800 MHz to 4.0 GHz under the turbo governor, which is left active in every campaign apart from one control campaign pinned at 2.4 GHz. The machine presents two NUMA nodes, node 0 comprising logical CPUs 0–23 together with their sibling threads 48–71 and node 1 comprising 24–47 together with 72–95, as Figure 5.2 shows.

Workloads and aggressors are deployed as Docker containers under Docker Engine 29.2.1 and pinned by `cgroup`, so that the placement of every process is fixed for the duration of a run and known to the analysis without having to be inferred from it. Counters are read with Linux `perf` and aggressors are generated with `stress-ng`.

5 Setup and Methodology

Machine: 2× Intel Xeon Gold 6442Y, 96 logical CPUs

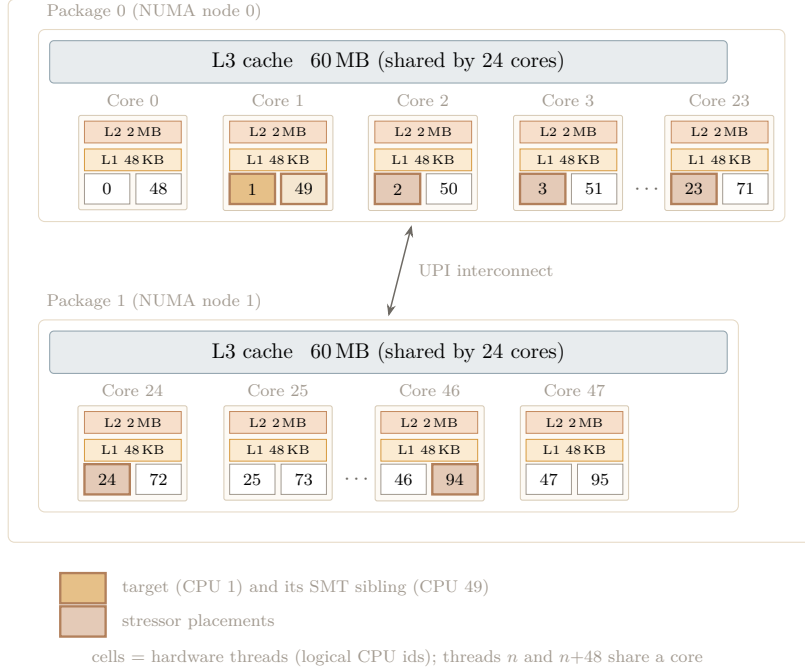


Figure 5.2: Topology of the testbed as reported by the operating system.

The placement code is fixed here and is the general contention format used for every experiment. The target is pinned to logical CPU 1 in every campaign, and a neighbour is placed at one of six positions relative to it, each of which exercises a different row of Table 2.1, as Table 5.1 sets out.

Table 5.1: Placement code: the target is pinned to logical CPU 1 throughout, while stressors exercise different resources.

Logical CPU	Position relative to the target	Levels shared
1	Same logical CPU	CPU time, and every level above it
49	SMT sibling of CPU 1	Pipeline, ports, ROB; L1, L2, TLB
2, 3	Adjacent cores, same socket	L3, mesh, DDIO, DRAM
23	Far core, same socket	L3, mesh, DDIO, DRAM
24	Remote socket	Inter-socket link
94	SMT thread, remote socket	Inter-socket link

The six positions are nested rather than disjoint, in that a neighbour on CPU 1 shares everything a neighbour on CPU 49 shares and adds the scheduler to it, while a neighbour on CPU 49 shares everything a neighbour on CPU 2 shares and adds the private caches and the pipeline. CPUs 2, 3 and 23 exercise the same row of Table 2.1 and differ only in their distance across the on-die interconnect, so that

retaining all three allows that distance to be tested rather than assumed, whereas CPUs 24 and 94 sit on the far socket and share, in principle, no more than the link between sockets and the power budget of the package.

5.2 Workload Progression

The measurement campaign uses three workloads in a progression in order to stress different aspects of resource usage.

The first is a proxy chain in which HAProxy operates as a layer 7 reverse proxy in HTTP mode in front of an nginx backend serving static files of 1, 2 and 5 MB, with the two functions in separate containers on separate logical CPUs. Since the traffic is memory-bound, two containers are involved rather than one, the proxy gets the request and serves by forwarding requests to the backend which returns a file, and the file size acts as a control that moves the amount of data touched per request without altering the code path. It is also the only campaign in which the key performance indicator is available per second on the client side, making it relevant for performance estimation in windows of time where system state changes and tenants get deployed in real time.

What the HAProxy implementation cannot do is vary the target's own memory behaviour independently, because its working set follows from the traffic offered to it and the cache level at which the function actually resides is much more difficult to tune without increasing traffic significantly. Network functions that have a very easily tuned memory footprint are suited to testing different contention levels, as they enable different topologies to test the impact of contention regardless of external factors not directly inherent to the function itself.

For this purpose a second measurement campaign focuses on memcached and redis. Both are key-value stores whose working set is determined by configuration rather than by traffic, and their value sizes and request rates are chosen by design so that the active working set resides in the first-level cache, in the last level cache, or in DRAM. The memcached campaign is a full factorial over three residence regimes, five contention scenarios and five iterations, giving 75 runs, while the redis campaign contributes 140 contention runs together with 15 control runs at pinned frequency, the latter existing to separate microarchitectural effects of SMT from the frequency lift associated with hyper-threading.

The traffic issued to workloads comes from a second host across the network interface, so that requests arrive at a rate set by the generator rather than by

the completion of previous requests and the packet path includes the interface, the driver and the kernel network stack. Different traffic generators are used across experiments, as some are better suited for different applications and functions.

5.3 Experimental Setup

Neighbours are synthesised with **stress-ng** in two modes. The CPU-bound mode executes arithmetic in a small loop, occupying issue slots and reorder buffer entries while touching almost no memory, whereas the memory-bound mode streams through a buffer sized to exceed the private caches, generating evictions in the last level cache and traffic at the controllers while leaving the pipeline comparatively idle. The two modes correspond to the two families of contention mechanism: structural occupancy and displacement or queueing.

In the memcached and redis campaigns the contention conditions are named and not numbered; their core placements are detailed in Table 5.2, which follows the format of Table 5.1.

Table 5.2: Scenario names used in the memcached and redis campaigns, mapped onto the placement code of Section 5.1.

Scenario name	Aggressor placement	Level exercised
solo	none	baseline
same_core_cfs	CPU 1	CPU time
smt_sibling	CPU 49	pipeline and private caches
cross_core_local	CPUs 2, 3, 23	L3 and memory controllers
both_smt_and_cross	CPU 49 with 2, 3, 23	both of the above

The experimental variable throughout measurements is where the neighbour runs rather than the intensity of its processing, since an orchestrator does not select the pressure a tenant applies to a shared resource but selects the host and the cores on which a container is placed, from which every subsequent interaction follows.

Synthetic aggressors are used in place of real functions for two reasons. They are controllable, in that the pressure they apply is set by configuration and repeats across runs, which is what allows placement to be varied while everything else is held constant, and each mode loads one shared resource at a time, which is what allows a signature observed in the counters to be attributed to a mechanism rather than to an unresolved mixture of them.

Counters are collected with **perf** at a cadence of one second, which sets the

temporal resolution of every further analysis and matches the resolution at which the key performance indicator is available. The events are requested as groups and time-multiplexed by the kernel, with each group scheduled onto the hardware for part of the interval and its partial count scaled to an interval estimate.

When the target function and the stressor share the same core counters have been extracted both on the core, measuring the full impact of the two functions and on the process itself in order to obtain a more complete view of the impact of the function itself on the counters, which is possible in every other case since the stressors do not pollute the count on the core but pollute the execution of the function.

Raw counts are not used as features directly, because a count over an interval depends as much on how much work the target managed in that interval as on how well it was executing, and contention impacts both aspects. Each count is therefore divided by a reference quantity: instructions, cycles, or elapsed time, and all of them are kept, since the choice of denominator determines which mechanism a feature is sensitive to. Every feature is additionally expressed as a ratio against a solo run of the same function on the same hardware, so that values from different workloads can be compared on a common scale.

5.4 KPI Measurement

Load is offered by a generator external to the function under test, and latency is recorded where a client would experience it rather than reconstructed from anything observed on the host. The proxy campaigns are driven by ApacheBench, which reports p99 latency once per second providing the per-second series, while memcached and redis are driven by memtier_benchmark, which reports p99 latency aggregated over the whole run. External traffic is generated by vegeta from the second host.

In every run the function is started and given a warmup period during which its caches fill and its allocator reaches a steady state, followed by a pre-stress window in which it serves traffic with no neighbour present and which supplies a within-run baseline. The aggressor is then deployed at its assigned placement, mid-run, and pressure is escalated through a sequence of phases rather than applied at full intensity at once, so that the transition between contention levels is observable and not only the endpoints, after which the run closes. Both streams carry epoch markers, the join between counters and key performance indicator is performed on those markers second by second, and warmup seconds are trimmed before any of it enters the analysis.

The within-run baseline permits one check that has to be made before the pre-stress window can be used as a reference. For each object size the p99 measured during the pre-stress window of a contention run equals the p99 measured across a dedicated solo run, at 25, 45 and 93 ms for the 1, 2 and 5 MB objects respectively, which establishes that the presence of the stressor container before it becomes active does not itself perturb the function.

5.5 Obtained Datasets

Table 5.3 states what the four campaigns produced.

Table 5.3: Campaigns, runs, key performance indicator granularity and records retained after trimming.

Campaign	Runs	KPI granularity	Records
HAProxy	57	per-second p99	12,762 per-second rows
memcached	225	per-run p99	75 aggregates
redis	420 + 45 control	per-run p99	140 aggregates
External traffic	50	per-second p99	7,033 + 2,130 rows

The asymmetry in the third column is the structural fact about this dataset, since counters are recorded per second in every campaign, which is why memcached contributes 75 aggregates, whereas the key performance indicator is recorded per second only for the two proxy campaigns. Any result requiring the two streams to be aligned in time is therefore available on the HAProxy and external-traffic data and nowhere else, while any result drawn from memcached or redis relates counters aggregated over a run to a single latency figure for that run.

6 Experiments

The aim of the experiment section is to verify and quantify the impact of contention. Every source of slowdown due to resource sharing is derived from theory, each of them is isolated and analysed to infer the correlation between the slowdown and the hardware behaviour.

The analysis follows two main variables and sweeps one as the other is fixed and known: functions with the same workload are placed in many different topologies to verify how positioning affects slowdown and functions of different loads are compared when deployed in the same topology. The observed quantity is the same in every experiment, the 99th percentile of client-side response latency over the measurement window, and it is always reported as a ratio against the solo runs of the same configuration, so that a value above one is a slowdown and a value below one is a speed-up. Every value quoted is a median over the runs of its cell. Latency is the only key performance indicator considered as results in the literature that attempt similar approaches always highlight that latency estimation is much more challenging and tends to be generally worse than throughput in terms of mean absolute error (MAE) [23].

6.1 Cross Topology Analysis

A competitor on the victim's own logical CPU shares everything, including the processor's time. Moving it to the SMT sibling returns the time but keeps the pipeline, the private caches and the TLB. Moving it to another core of the same socket returns those as well and keeps only the last level cache and the path to memory. Moving it to the remote socket keeps the interconnect and the power budget and nothing else. Since each step outwards withdraws a mechanism and adds none, the cost of a neighbour should fall monotonically with distance, and it should fall in steps rather than smoothly, one step per level withdrawn.

The measurement is a placement sweep. HAProxy runs in HTTP mode pinned to logical CPU 1 and traffic is offered from a second machine over the same subnet.

6 Experiments

A single **stress-ng** container is then pinned to one position at a time: the victim’s own logical CPU, its SMT sibling, the backend’s core, a further core sharing the same last level cache, and two positions on the remote socket. Both pressure modes are used, random cache eviction and memory copy, and each cell is run three times. Nothing else changes between runs, so the only variable is where the competitor sits.

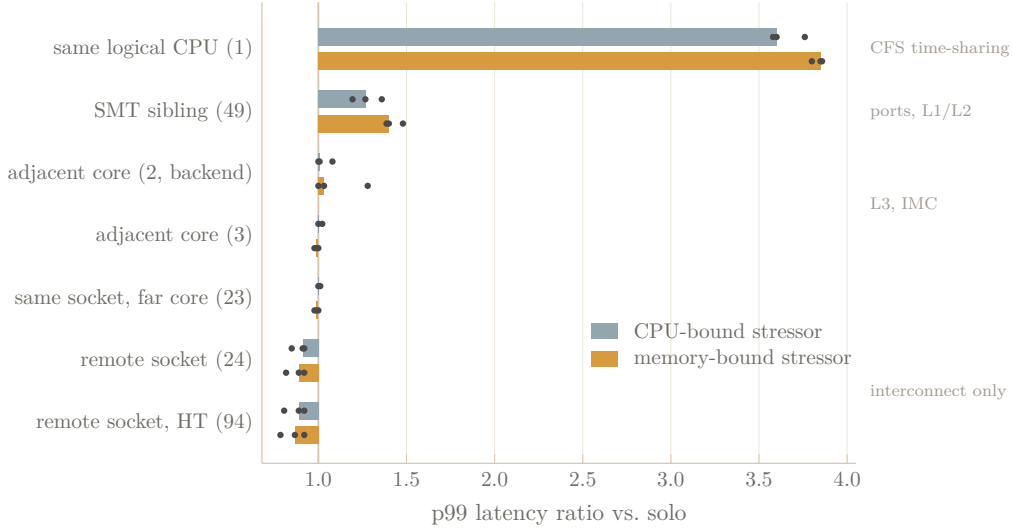


Figure 6.1: P99 response latency of the HAProxy victim, normalised against its solo baseline.

Figure 6.1 presents the sweep results. At first glance, it is clear that memory-bound stress has a greater impact than CPU-bound stress. A competitor on the same logical CPU costs a factor of 3.6 to 3.8. On the SMT sibling it costs a factor of 1.3 to 1.4. From the adjacent core outwards it costs approximately one. The file size that the proxy is serving is small enough to live mostly in L1 and L2 cache so a stressor putting pressure on L3 or DRAM doesn’t impact the proxy execution.

Although intuitively the difference between same core and SMT should be a factor of 2, the observed factor is closer to 3 as latency is a tail quantity and the sharing is not continuous: a request that arrives while the stressor holds the CPU does not proceed at half speed, it waits for the remainder of the scheduler’s slice. Time-sliced sharing therefore hurts a latency percentile by more than it hurts throughput, and the gap grows with the length of the slice.

The cross-node topology case highlights a counterintuitive behaviour as latency slightly drops when stressors are deployed on the same machine but on different NUMA-nodes. Latency is consistently below the solo baseline, between 0.87 and 0.92, across both stressor types and all runs. A competitor that shares no microar-

chitectural structure with the victim makes it slightly faster.

6.2 Variable Frequency Effects

Frequency is mentioned alongside the caches and the memory controllers as a shared resource, with the package as its set of neighbours and the power budget as its mechanism. A busy neighbour consumes power budget and the target’s clock falls. That reading is correct in a regime where all the cores are near saturation. It is indeed a desired regime for a provider, since it implies that every core is being used at the maximum, but it is not the only possible regime a service runs in, making frequency variations relevant and potentially beneficial.

In order to quantify the impact of frequency variations on the execution of functions, the experiments carried out analyse the case where one or more processes are running on sibling cores and the frequency is first allowed to vary and then fixed. Redis is the chosen network function and it serves 16 KiB values from a container pinned to logical CPU 1, at a fixed offered rate, with the KPI taken over a sixty second window. Two scenarios are compared, solo and a stressor on the SMT sibling, and the pair is run twice: once under the default `intel_pstate` governor with turbo enabled, and once with every logical CPU pinned at 2.4 GHz and turbo disabled through `no_turbo`. If the frequency is enhancing the function execution, the effect is expected to disappear when the clock is held constant.

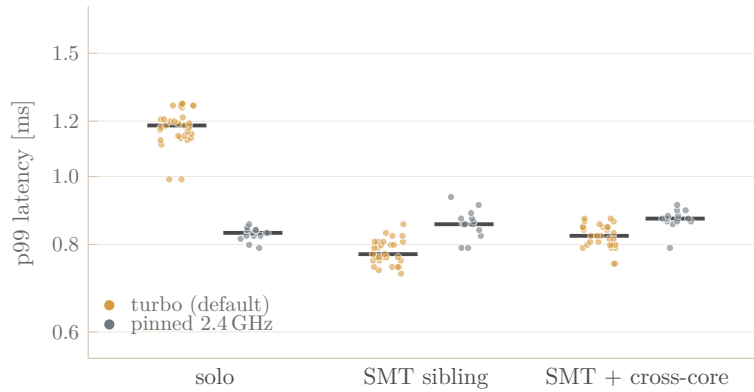


Figure 6.2: Redis p99 latency under SMT-sibling contention, with the default turbo governor and with all cores pinned at 2.4 GHz.

Under the default governor, SMT-sibling contention improves the KPI by 34 %, from 1.18 ms solo to 0.78 ms with the sibling loaded, a separation the Mann-Whitney

6 Experiments

test places at $p \approx 5 \times 10^{-10}$. With the cores pinned the improvement is gone and the sign reverses to +3 %, which is the direction theory suggests.

The mechanism is visible in the effective frequency. A Redis process at moderate rate spends most of its time blocked in `epoll`, and the governor, seeing a core that is mostly idle, scales the frequency down to 1.2 GHz. A stressor on the SMT sibling makes the physical core continuously busy, and the governor clocks it up to 3.0 GHz. The IPC penalty from sharing the pipeline is present throughout and is not small, about a third in both settings, but at this utilisation a factor of 2.5 in clock rate is larger than a third of the work per cycle, and the KPI is enhanced as frequency rises.

The second observation in Figure 6.2 is the same mechanism read from the other side. Pinned-solo, at 0.83 ms, is faster than turbo-solo at 1.18 ms. Nothing is contending in either case. The difference is that the default governor is allowed to let a lightly loaded core idle down, and the pinned configuration is not. A service of this shape is therefore slower alone under the default policy than it is with a busy neighbour, and slower again than it would be if the clock were simply held fixed.

This has a consequence for every ratio reported in this chapter and for any measurement of the same nature. The solo baseline is not a fixed reference on a machine with default power management, because the governor sets the target's clock from the target's own utilisation. A contention ratio measured under such a governor is a comparison of two different clock regimes, and the fingerprint associated with contention must take into account also frequency at any given moment.

Frequency proves to be unique as it does not behave like other shared resources: raising the load on the functions may prompt the governor to raise the clock frequency thus making the performance enhance. This behaviour explains why at the remote socket, where no structure that could remove cycles from the victim is shared, a change in the KPI can only come from a change in the rate at which cycles are delivered, namely frequency.

6.3 Cross Workloads Analysis

The idea of decomposing a process into contentiousness and sensitivity regarding shared resources describes degradation as a property of a pair rather than the workload by itself, assessing a contention scenario means taking into account both the target and the aggressor. The prediction under test is the strong form of that claim. In order to explore how contentiousness and sensitivity can vary on the same func-

tion, target and aggressors are fixed while varying workloads examine how impactful a given aggressor is for different memory fingerprints of the target process.

The measurement uses a memcached function, whose working set per unit time is moved through the memory hierarchy by changing the value size and the offered rate. The average throughput in this experiment varies between 3 and 4 Gbps. Three regimes are configured: the L1 regime runs at an effective 1.33 GHz with a last level miss rate of 3 %, the L3 regime at 2.24 GHz with the same miss rate, and the DRAM regime at 3.0 GHz with 71 %. Four contention scenarios are applied to each: a stressor on the victim’s own logical CPU under CFS, a stressor on the SMT sibling, stressors on other cores of the same socket, and the last two together. Each cell is normalised against the solo baseline of its own regime.

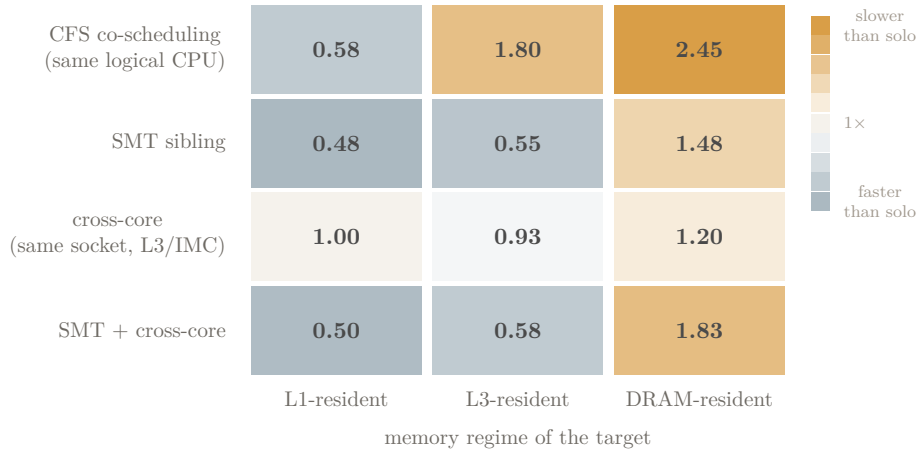


Figure 6.3: Memcached p99 latency ratio against the solo baseline of the same regime

Figure 6.3 shows in each row the progression of the three different workloads under the same aggressor’s position, thus how sensitive the target is to the chosen stressor. One stressor on the SMT sibling gives a ratio of 0.48 when the target is L1-resident, 0.55 when it is L3-resident and 1.48 when it is DRAM-bound. Co-scheduling on the target’s own logical CPU gives 0.58, 1.80 and 2.45 across the same three regimes. Neither the aggressor nor its placement changes along a row. Only the target does, and between the ends of the first row the effect changes sign and triples in magnitude.

The mechanism is due to frequency scaling: an L1-resident memcached idles between requests and is clocked down, so it has headroom that a busy sibling converts into speed, while a DRAM-bound memcached is already held at maximum turbo, because a core stalled on memory is busy as far as the governor is concerned, so there is no headroom left and the IPC penalty arrives at the KPI untouched.

6 Experiments

The effect of Simultaneous Multithreading is evident in the figure. The SMT row changes sign across the regimes, because its only cost is work per cycle and that cost is paid out of whatever clock headroom the target has. The same-logical-CPU row rises monotonically with regime severity, from a speed-up at L1 to 2.45 at DRAM, because time-slicing removes cycles outright and the headroom can offset it only while the victim is idle enough for the headroom to be large.

The counter fingerprint does not flip when the KPI does. The SMT stressor removes 32 to 34% of the victim’s IPC in every regime, consistent to within a percentage point, and it does so in the two regimes where p99 improves exactly as in the one where p99 degrades. The architectural event and the service-level consequence are therefore separate facts, and the mapping from the first to the second depends on a property of the target that no single counter variation reports. A monitor that reads a 33% IPC drop and infers a violation will come to the wrong conclusion for L1 and L3 residency. It follows that counters can be mapped to a KPI only once the target’s regime is known, which highlights the need for classification models that may provide this mapping.

As expected cross-core interference is negligible until the working set leaves the private caches. The stressors on other cores of the same socket give ratios of 1.00 at L1 and 0.93 at L3 against 1.20 at DRAM, and the same scenario against Redis, which is L1-resident at every value size tested, gives 1.01 to 1.02. Cross-core placements stress memory controllers and are thus impactful only when a function depends actively on the DRAM controller and memory pages.

The combined scenario at DRAM, SMT sibling and cross-core stressors together, is bimodal across iterations: three runs land near 4.3 ms and two at 21 to 26 ms, with nothing in between. The condition sits at the point where the memory controllers saturate, and saturation is a threshold rather than a slope, so the two groups are two outcomes of the same configuration rather than a spread around a mean. It is reported as two modes, since a median would describe neither.

6.4 Findings

Table 6.1 takes the levels of Table 2.1 one at a time and sets each prediction against what was measured. Four verdicts are used: *confirmed*, *conditional* where the prediction holds only for some targets, *amended* where the mechanism is real but its stated effect is not, and *not observed*.

Every number in this chapter was obtained with the KPI in hand. The victim

Table 6.1: Predictions of Table 2.1 against the measurements of this chapter.

Resource	Shared with	Predicted	Observed	Verdict
CPU time	Logical CPU	Time-slicing	3.6–3.8× proxy; 0.58–2.45× memcached	Confirmed
Pipeline, ports	SMT sibling	Occupancy	IPC –32 to –34 %, every victim	Confirmed
L1, L2, TLB	SMT sibling	Displacement	1.3–1.4× proxy; 0.48–1.48× memcached	Amended
L3, mesh, DDIO	Same socket	Eviction	1.00× at L1, 0.93× at L3	Conditional
Memory controllers	Same socket	Queueing	1.20×, DRAM regime only; bimodal with SMT	Confirmed
Frequency	Package	Power budget	–34 % under turbo, +3 % pinned	Amended
Inter-socket link	Remote socket	Remote latency	0.87–0.92×	Not observed

was instrumented, the load generator reported its own latency, and each counter reading could be checked against a measured outcome.

7 Leveraging Data

During experiments the KPIs of the example function were always given and readable. The function was instrumented, the load generator reported the latency it measured, and each counter reading could be checked against an outcome that was already known. An operator hosting a function it does not own is never in that position, since the container is opaque, the client is somewhere else, and the only stream that reaches the host without asking the tenant for anything is the counter stream.

To assess quality of service a provider needs to know how much latency the tenant is experiencing, which part of the hardware is working the hardest in the infrastructure, since that is what decides whether a given neighbour can hurt it and which topologies are acceptable. This chapter builds estimation tools using the data to provide examples as to how PMUs can serve in inferring performance and ensuring QoS.

7.1 Feature Construction and Validation

The unit of input is one row per second per run, and four families of features are built on it.

The first family is the counter set itself, normalised in three ways: per second, per instruction, and per cycle. The second is the set of derived ratios that the normalisation makes interpretable, among them instructions per cycle, the private and shared cache hit rates, and the stall fractions of the pipeline decomposition. The third is temporal context, built as a trailing rolling mean and a trailing rolling standard deviation over a fixed window, together with the lag-1 value of every counter. All three of these are causal, in the sense that a value computed for second t uses only seconds up to and including t , so a streaming observer could compute them online without knowing the future of the run.

The fourth family is derived by comparing existing features to the baseline counters. Each counter is divided by the median of that same counter over the pre-stress

window of its own run, which is the interval in which the function serves traffic alone. The resulting feature does not state how many misses per instruction the function is taking, it states how many it is taking relative to how many it took when it was alone. This requires offline profiling of functions by placing them on known isolated hardware and moving them after a period of observation.

Seconds drawn from the same run are not independent samples since every run is unique, set by object size, offered load, and whichever clock regime the governor settled into, and consecutive seconds inside a run differ from each other by very little. Under plain five-fold cross-validation, where rows are shuffled and split at random, almost every held-out second has near neighbours from its own run sitting in the training set. Thus training models on the same static feature set scores $R^2 = 0.954$. When the folds are formed by run, so that no second of a test run appears anywhere in training, the same features score $R^2 = 0.737$. To ensure that the model has no training bias the folds are formed by run.

Gradient-boosted regression trees are used in order to carry out timeframe estimation, because the features are heterogeneous in scale, correlated with each other, and related to the target through thresholds rather than through smooth global trends.

7.2 Per-Second KPI Estimation

The HAProxy measurement campaign is used in order to estimate timeframes since the load generator reports p99 once per second making it possible to compare the measured and predicted timeseries. Figure 7.1 shows predicted against measured latency on held-out runs, and Figure 7.2 shows the same model reading a single run in time.

The ablation is the result, and it takes four numbers. Static counters alone reach $R^2 = 0.737$. Adding the causal temporal context raises this to 0.810. Adding the baseline-relative features and predicting the logarithm of the target raises it to 0.850. The blend of the two model families reaches $R^2 = 0.873$.

What the baseline-relative features fix is run-to-run level shift. Two runs sitting at the same contention intensity can be at very different absolute latencies, because payload size and offered load differ between them, and a model given only absolute counters struggles to infer the level and the deviation from the baseline. Dividing by the run's own pre-stress median removes the sheer magnitude and moves the inference towards how far latency at a given second has moved from what the function's

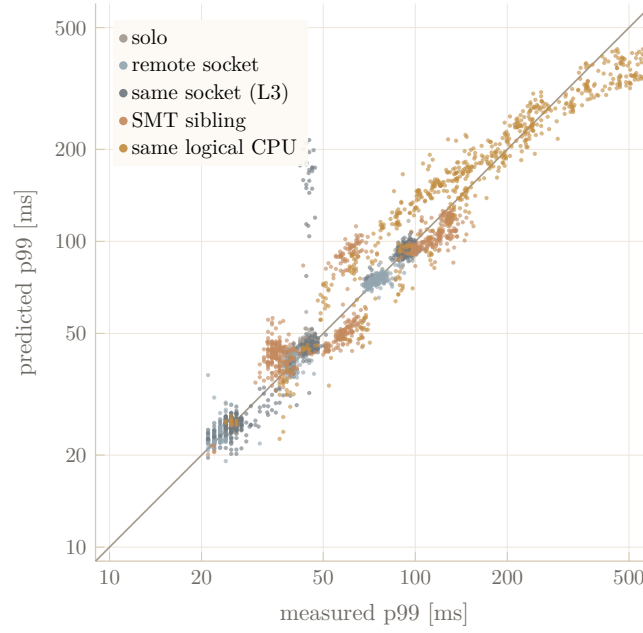


Figure 7.1: Predicted against measured per-second p99 on HAProxy.

baseline performance. Predicting on a logarithmic scale helps the model predict the magnitude of the latency rather than its exact value, which increases the accuracy when p99 spans more than an order of magnitude across the measurements.

The aggregate hides a strong dependence on placement, and Table 7.1 breaks the aggregate down by placement.

Table 7.1: Out-of-fold R^2 of the per-second p99 estimator, by placement of the aggressor.

Placement	R^2
Solo, no aggressor	≈ 0.99
Remote socket	≈ 0.99
Same logical CPU	0.85
Same socket, shared L3	0.13

The coefficient of determination is a ratio against the variance present in the subset it is computed on, so the ordering in the table reflects how well the model tracks a placement and how much information it holds on a specific placement. The same-socket L3 row clearly carries little information, since that is the placement where the counters barely move, and a target that does not vary offers nothing to explain while scoring every small residual against a very small denominator.

7 Leveraging Data

Figure 7.2 demonstrates how the regressor can predict varying latency over time. On runs held out in their entirety the estimate follows the escalation staircase, stepping up at phase boundaries it was never shown, and on a solo run it holds the level instead of being biased by degradation, which is the most common scenario in the sample runs used for training.

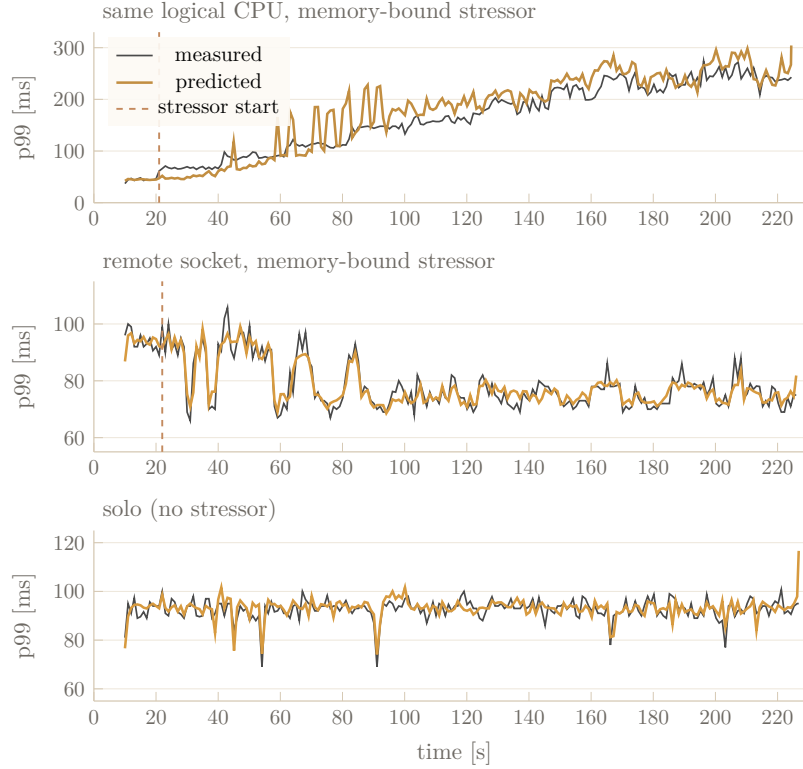


Figure 7.2: Measured and estimated per-second p99 on held-out HAProxy runs.

The external-traffic campaign gives four readings on how far this transfers. Out-of-fold on that campaign the estimator reaches $R^2 = 0.907$. A single model spanning both the 1 KB and the 1 MB workloads, with no feature telling it which one it is looking at, reaches 0.957, so the mapping from counters to latency does not need to be told the payload size. Applied zero-shot to a stressor type absent from training it reaches 0.87, so the estimator is not memorising the signature of a particular aggressor. Trained on light workloads and tested on heavy, however, it collapses to 0.02, and the errors are systematic underestimates rather than noise. This model cannot generalise intensity, since it has learned a mapping over the range of pressure it was shown, and estimates for pressure outside that specific range remains biased.

7.3 Aggressor Localization

Reducing aggression from other tenants is key to optimising process deployment and in order to do so it is essential to know where the cause of slowdown is placed in the hardware. Given the HAProxy measurements and target-side counters alone, the trained classifier may identify the placement of the aggressor among five possibilities: none, same logical CPU, SMT sibling, same socket sharing L3, and remote socket. Figure 7.3 reports the confusion matrix.

actual placement	remote socket	3	80	17		
	same socket (L3/IMC)	4	7	89		
	SMT sibling				100	
	same logical CPU			2		98
		no stressor	remote socket	same socket (L3/IMC)	SMT sibling	same logical CPU
		predicted				

Figure 7.3: Per-second localization of the aggressor from target-side counters.

Five-way accuracy is 83.3 % against a majority baseline of 37.9 %, that being the accuracy obtainable by always answering the most frequent class. The first question the model answers is whether there is a core-level aggressor at all, and that is answered at 99.8 %. One line of ablation separates the contributions: static counters give 76.8 %, and adding fifteen seconds of trailing context gives 83.3 %. Giving context to the model raises the F1 score of the no-stressor class from 0.18 to 0.49. The no-stressor case is critical for classification as a single second under a mild aggressor looks much like a single quiet second and to distinguish the two a model needs a window of time long enough to realise that the counters barely change. What the classifier confuses with no stressor the most is the same-socket L3 placement, which is the placement that is most similar to leaving the function alone.

The same model and task trained and tested within memcached reaches 83 %, so the approach is not specific to a proxy. A localizer trained on HAProxy and applied to memcached fails to generalize and cannot accurately classify. Topology signatures

are therefore a property of the pair formed by the workload and the placement, not of the placement alone, and an operator deploying this would have to profile each function it intends to observe.

7.4 Regime Classification

The regime decides the sign of the interference effect. A scheduler that knows the placement but not the tenant’s residency predicts the direction of the p99 change wrongly. The provider sets the placement but cannot read the tenant’s working set, so the regime has to be estimated.

Table 7.2 describes how the classifier performs across different scenarios. On memcached, with the payload tuned to L1, L3 or DRAM residency, classification reaches 100 % on the held-out split and under leave-one-out, for logistic regression, random forests and gradient-boosted trees. The test split is six runs, the classes are separable by construction, and 41 of the 55 two-feature pairs score the same, so the model is expected to separate classes effectively. Recursive feature elimination selects instructions per cycle and effective frequency since it is the most sensitive to contention at 33 %, against 71 % for L1 replacements with dTLB misses per thousand instructions.

Table 7.2: Regime accuracy with and without contention in the training set. Core-level pools the same-CPU and SMT placements.

Test set	Training	Solo	Cross-core	Core-level
memcached	solo only	1.00	1.00	0.33–0.47
	all conditions	1.00	1.00	0.93–1.00
redis (unseen)	solo only	1.00	1.00	0.00–1.00
	all conditions	1.00	1.00	1.00

Redis, absent from training, is used to test whether the model generalises to an unknown workload. Seventy redis runs serve 1 KB values and are L1-resident. Contention-augmented logistic regression places 70 of 70 correctly under every condition. The tree ensembles match it except under an SMT sibling, where they fall to 0.10 and 0.30, because they split on absolute thresholds fitted to memcached magnitudes that redis does not share.

7.5 Estimation Layer

One counter stream, read once a second on the host, feeds a regressor that returns the estimated indicator, a classifier that returns the estimated regime, and a classifier that returns the estimated source of interference. Taken together they are the estimation layer of a digital twin of a containerized network function, which is the part of such a twin that keeps a model of the tenant’s state current from observation alone. Figure 7.4 shows the possible pipeline with the components that would complete a twin, a forward model able to compute different scenarios and a loop able to act, shown outside the boundary of this work.

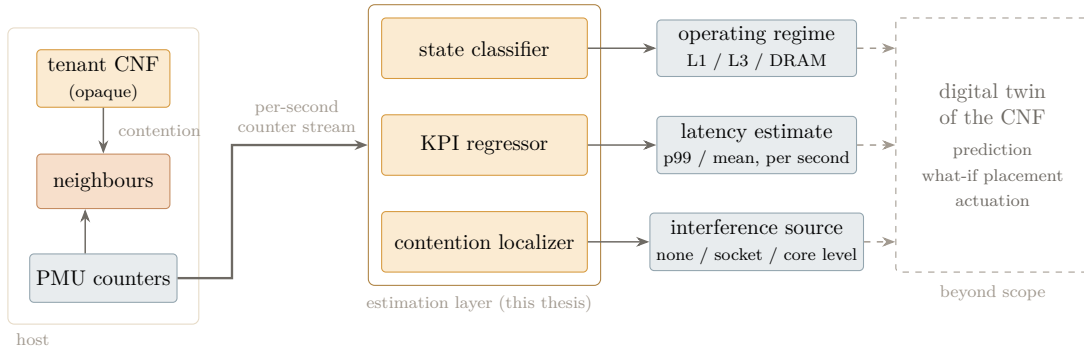


Figure 7.4: The estimation layer: one counter stream, three estimators.

What has been demonstrated is per-second, host-side, tenant-agnostic estimation of all three quantities on real containerized network functions. The models are offline, trained after the campaigns closed rather than updated as data arrives. The measurements come from a single microarchitecture. The aggressors are synthetic, chosen to isolate one sharing mechanism at a time rather than to resemble a real neighbour and profiling is per workload.

An estimate of the regime and of the current source of interference is what a conditional placement estimation needs as its starting state, since knowing whether a candidate neighbour would hurt a target function comes from knowing whether the two functions contend for a structure the target is actually using. An estimate of the indicator, produced without accessing the tenant’s software, is what turns a quality of service target into something the platform can check for itself. A scheduler would read the estimated p99 of the functions it already hosts, admit a new one only where the indicators are predicted to keep meeting their target, and move it once they degrade.

8 Conclusions

The campaigns conducted in this work show that the trace a containerized function leaves in the hardware carries information that can describe the function itself, even when the provider has no access to the software it is running. From a single counter stream, read once a second on the host and without any cooperation from the tenant, three quantities were estimated: the tail latency a client is experiencing, the level of the memory hierarchy the function is actually working in, and the position of the process interfering with it. The estimates hold on real network functions and at the granularity at which a service changes, following the escalation of contention within runs the model had never seen. The measurements also show that the mapping between the two levels is not a simple monotone relationship between pressure and slowdown. Frequency scaling can invert the sign of the effect, so that a busy neighbour on a sibling thread improves a lightly loaded function instead of degrading it, and the same aggressor placed in the same position produces opposite outcomes depending on the memory regime of the target, while the counters report the same loss of instructions per cycle in both cases making a learned mapping necessary to keep track of the real performance.

In order to build a real digital twin that can also optimise scheduling and interact with the real environment both estimation and decision making is needed, thus a feedback loop. The thesis demonstrates the forward direction of the loop, meaning the part that observes the infrastructure and keeps an estimate of the state of a tenant current. A digital twin requires two further components that remain outside the scope of this work: a forward model able to answer what would happen if a given function were placed next to another one, and an actuation path that acts on that answer and observes the outcome, closing the feedback.

Strengthening the forward half is also necessary. The neighbours used here are synthetic and were chosen to load one shared resource at a time, which is what allows a signature to be attributed to a mechanism, but a real host is populated by other network functions, each contentious and sensitive in its own way, so campaigns opposing containerized functions to one another are the natural continuation.

8 *Conclusions*

The measurements come from a single dual-socket machine with fixed placements, whereas an orchestrator moves workloads across a cluster of heterogeneous hosts and changes their neighbours over time. The models themselves are offline and profiled per function, and an operator running this continuously would need them to be updated as data arrives. The thesis shows that a strong correlation exists and that the proposed methods can both tackle the estimation of performance as well as recognising topologies and interference in non-isolated environments.

Acknowledgements

This work would not have been possible without the guidance of Prof. Davide Borsatti, whom I sincerely thank for his time, support, and friendship. I would also like to thank Prof. Leonardo Linguaglossa, who gave me the opportunity to carry out my research in Paris under his supervision and who has always gone above and beyond to help me navigate the bureaucratic challenges along the way. My heartfelt thanks also go to Prof. Walter Cerroni, who has always believed in my abilities and made this collaboration possible.

I cannot fail to mention all my dear friends and all those who care for me; every step of this journey has been made more meaningful by having you by my side. This achievement was only possible thanks to Denise, my rock, who kept me steady and grounded even on the most treacherous terrain. This thesis is dedicated to my mother, who fought every day for twenty-four years so that I could become the person I am today, and to my father, who embodies the purest and deepest essence of who I am: resolute, indomitable, and free.

Bibliography

- [1] Danielle E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, March 2016. USENIX Association.
- [2] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Cambridge, MA, 6 edition, 2017.
- [3] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014.
- [4] Charles Shelbourne, Leonardo Linguaglossa, Tianzhu Zhang, and Aldo Lipani. Inference of virtual network functions’ state via analysis of the cpu behavior. In *2021 33rd International Teletraffic Congress (ITC-33)*, pages 1–9, 2021.
- [5] What is Multithreading? <https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html>, 2026. [Online; accessed 18-August-2026].
- [6] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, September 1982.
- [7] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. Contention-aware performance prediction for virtualized network functions. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM ’20*, page 270–282, New York, NY, USA, 2020. Association for Computing Machinery.

Bibliography

- [8] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostić. Re-examining direct cache access to optimize i/o intensive applications for multi-hundred-gigabit networks. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.
- [9] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward predictable performance in software packet-processing platforms. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 11, USA, 2012. USENIX Association.
- [10] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model: quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 62–75, New York, NY, USA, 2015. Association for Computing Machinery.
- [11] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, page 12–21, New York, NY, USA, 2011. Association for Computing Machinery.
- [12] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 639–650, 2013.
- [13] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.
- [14] NFV White Paper. Network functions virtualisation: An introduction, benefits, enablers, challenges i& call for action. issue 1. October 2012.
- [15] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon ec2 data center. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, 2010.

- [16] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. *SIGOPS Oper. Syst. Rev.*, 42(2):26–35, March 2008.
- [17] Yaozu Dong, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–10, 2010.
- [18] Florian Wiedner, Max Helm, Sebastian Gallenmüller, and Georg Carle. Hvnet: Hardware-assisted virtual networking on a single physical host. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6, 2022.
- [19] Lizhuang Tan, Wei Su, Wei Zhang, Jianhui Lv, Zhenyi Zhang, Jingying Miao, Xiaoxi Liu, and Na Li. In-band network telemetry: A survey. *Computer Networks*, 186:107763, 2021.
- [20] Priyanka Naik, Dilip Kumar Shaw, and Mythili Vutukuru. Nfvperf: Online performance monitoring and bottleneck detection for nfv. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 154–160, 2016.
- [21] Nguyen Van Tu, Jae-Hyoung Yoo, and James Won-Ki Hong. Pptmon: Real-time and fine-grained packet processing time monitoring in virtual network functions. *IEEE Transactions on Network and Service Management*, 18(4):4324–4336, 2021.
- [22] Masahiro Dodare, Yuki Taguchi, Ryota Kawashima, Hiroki Nakayama, Tsunemasa Hayashi, and Hiroshi Matsuo. Nfv-vipp: Catching internal figures of packet processing for accelerating development and operations of nfv-nodes. In *2019 15th International Conference on Network and Service Management (CNSM)*, pages 1–4, 2019.
- [23] Qiong Liu, Tianzhu Zhang, and Leonardo Linguaglossa. Non-invasive performance prediction of high-speed softwarized network services with limited knowledge. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*, pages 2328–2337, 2024.

Bibliography

- [24] Todd Mytkowicz, Peter Sweeney, Matthias Hauswirth, and Amer Diwan. Observer effect and measurement bias in performance analysis. 08 2010.
- [25] Denis Bakhvalov. *PERFORMANCE ANALYSIS AND TUNING ON MODERN CPUS*. "Self-published", 2020.
- [26] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*, 2026. Order Number: 253669.