
KubeCon
CloudNativeCon
North America 2024

Hi everybody. Today I want to talk about **memory noisy neighbor**.

Run 20%-50%
more transactions

Reduce tail latency
by 20-80%

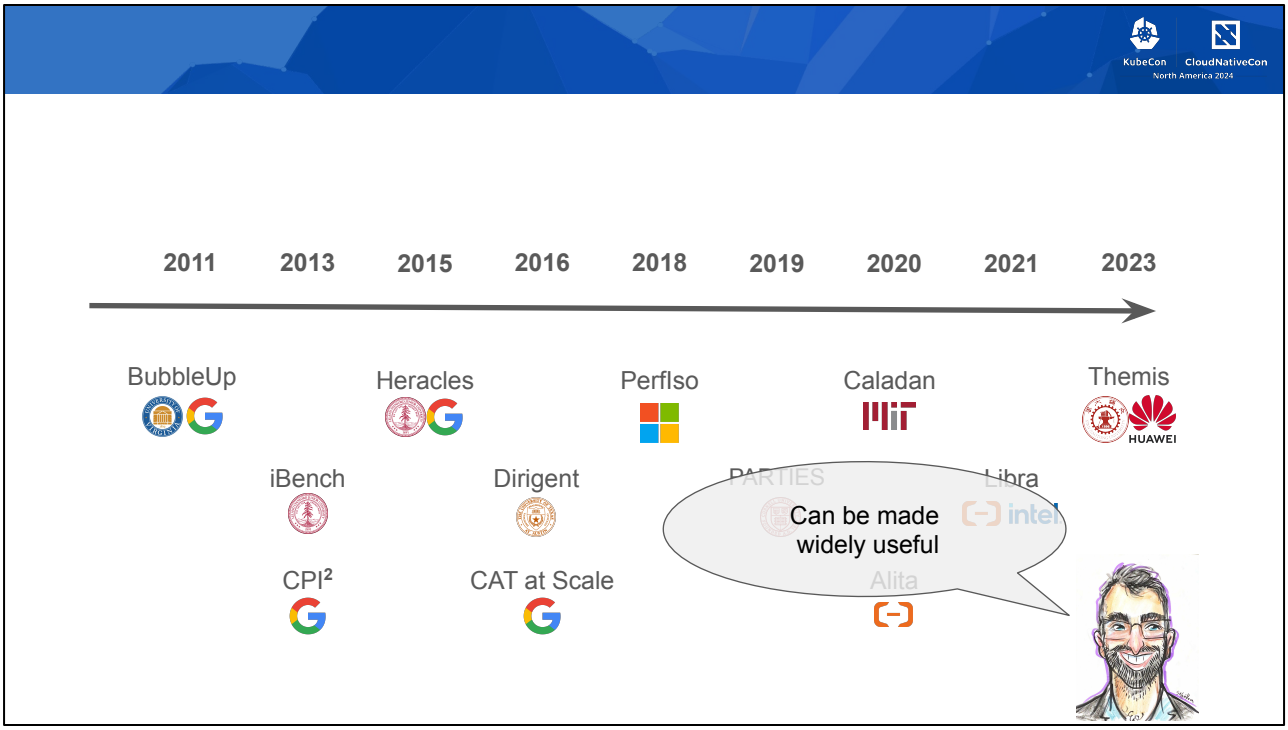
What if I told you this story that a small group of engineers got together and built a **secret** capability that allows them to run their workloads:

- **20 to 50% more efficiently** on less hardware
- and improves **tail latency substantially** by factors of 4, 5, 13?

That would be fantastic, right? *How can we not know about this?*

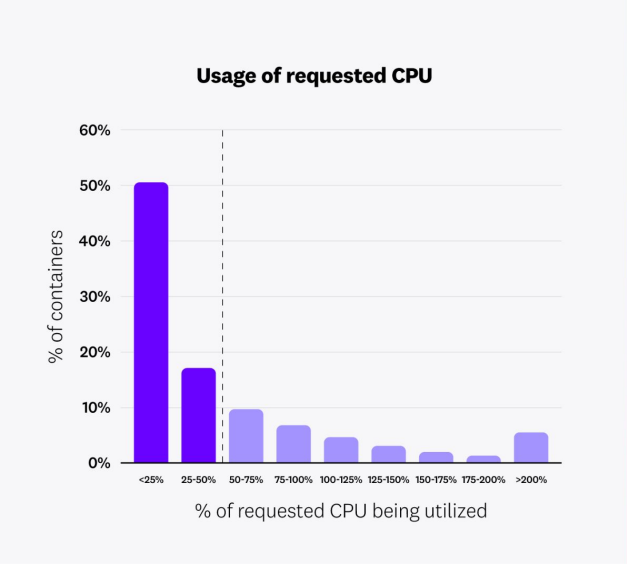
[pause]

Well, it turns out this capability seems to actually exist, although the development was not in secret.



There's been papers over the last more than a decade by top universities and well-known hyperscalers that explores this capability.

Today I want to give an overview of what's known about this. The reason I'm giving this talk is that I believe the knowledge that is out there is **ready to be made into practical systems** that can benefit the entire Kubernetes community.

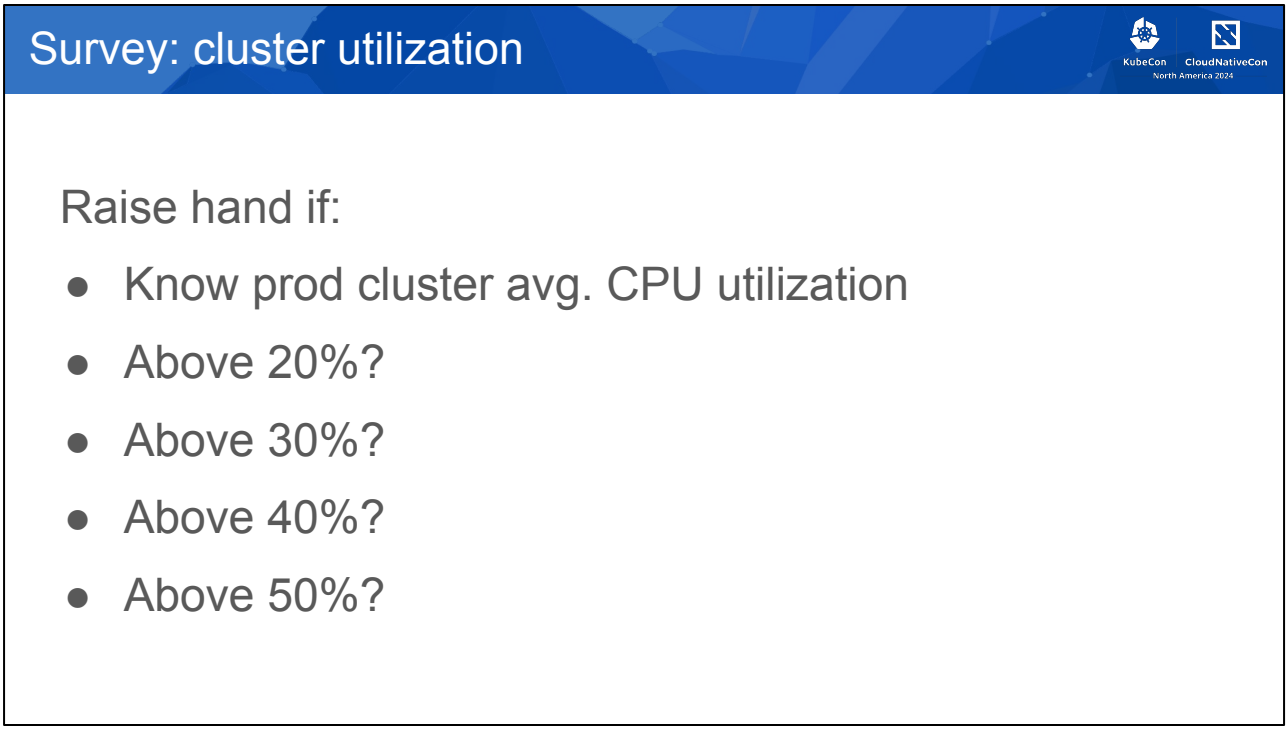


>65% OF CONTAINERS USE LESS THAN HALF OF REQUESTED CPU

Source: Datadog

source

You might know one of these surveys. This is one published by Datadog showing that containers use a lot less CPU than what we request for them.



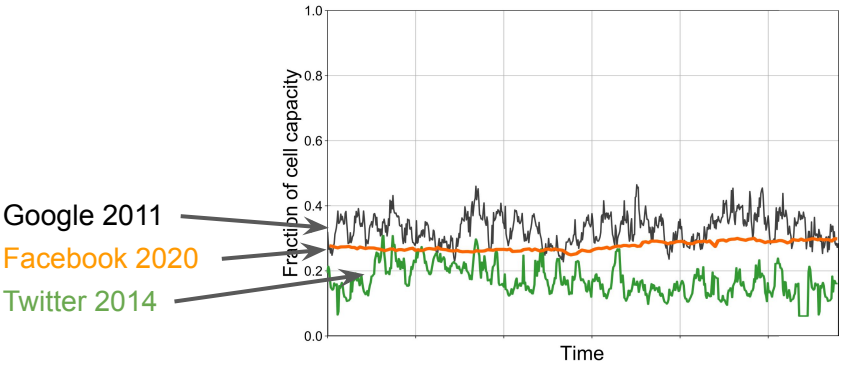
And so I want to perform a similar survey here today about cluster utilization.

[Interactive moment]

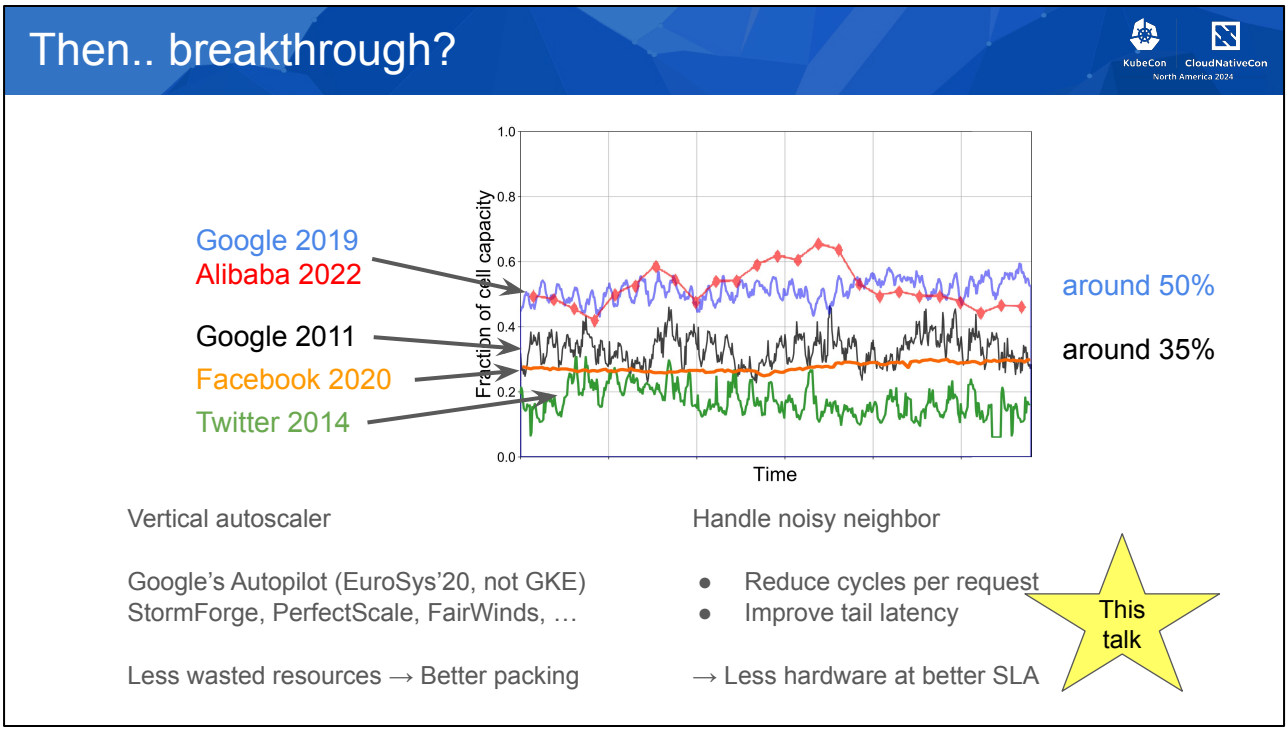
I'm going to ask you to please raise your hands if you know what an average CPU utilization is for your production clusters serving user-facing traffic.

Now leave your hand up if that cluster utilization is above **20%**... Leave your hand up if it's above **30%**... Above **40%**...?**50%**...?

Hyperscaler CPU was low…



Well, if you said between 20 and 40%, *you're in good company*. Some of the **largest companies on the planet**, with large engineering teams able to optimize their deployments, have published their average CPU utilizations and have results between 20 and 40%. You can see Google, Facebook and Twitter.



But what seems to have happened over the last few years is that some companies have been able to increase their efficiency **quite substantially**. As Google was around 35% in 2011, they published around 50% CPU utilization in 2019.

So what happened to increase efficiency so much at these organizations?

[pause]

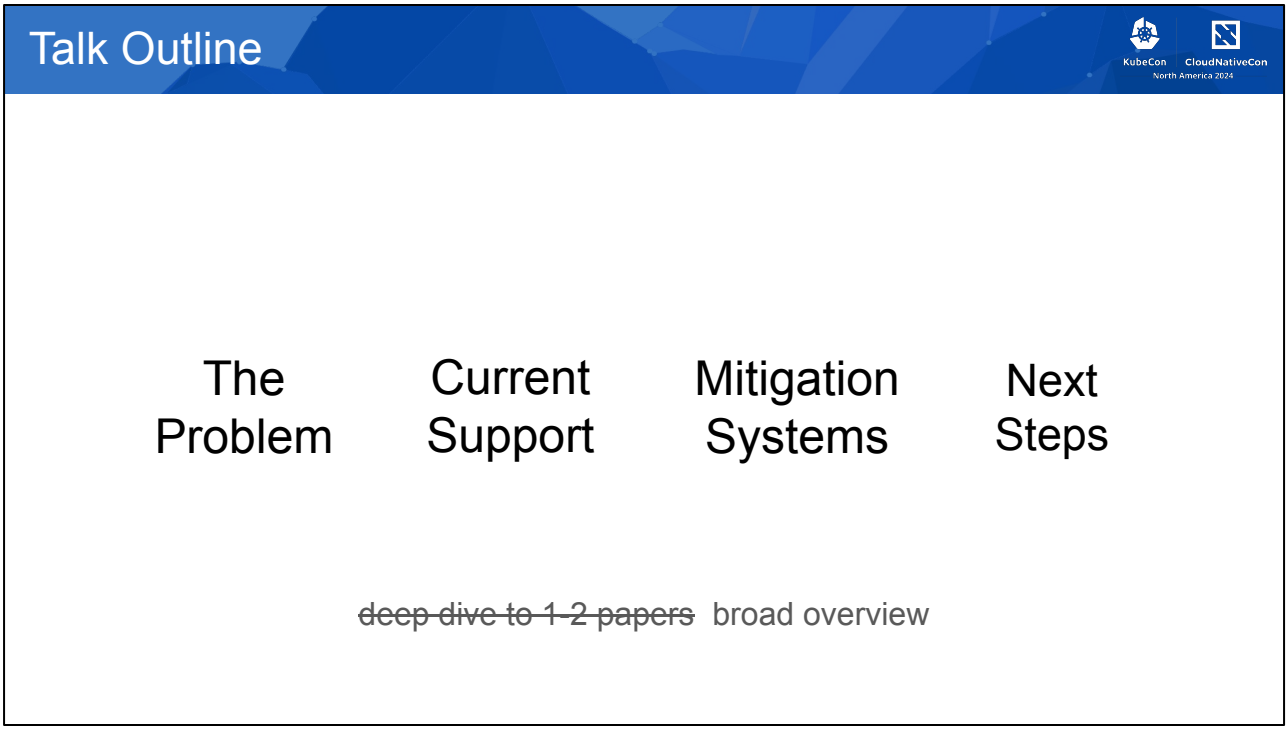
Well, reading published work, there seems to be **at least 2 contributors** that I was able to find:

1. Advancements in **vertical auto scaling**
    - There's a very interesting Google paper called Autopilot, published in Eurosys 2020
    - And this is *not* the GKE autopilot
    - This is a system used internally in Borg
    - Where they published how Google adjusts requests for memory and CPU to pack workloads tighter together
    - And there are companies that do this for the Kubernetes community today like Stormforge, Perfect Scale, and others

[pause]

2. The second contributor, which is going to be the **topic of this talk**, is the company's ability to handle noisy neighbor

- resources than they should
- Handling this type of behavior allows:
  - A reduction in the number of cycles used to process requests
  - And improvement in latency
- And I'll cover those in the next few slides

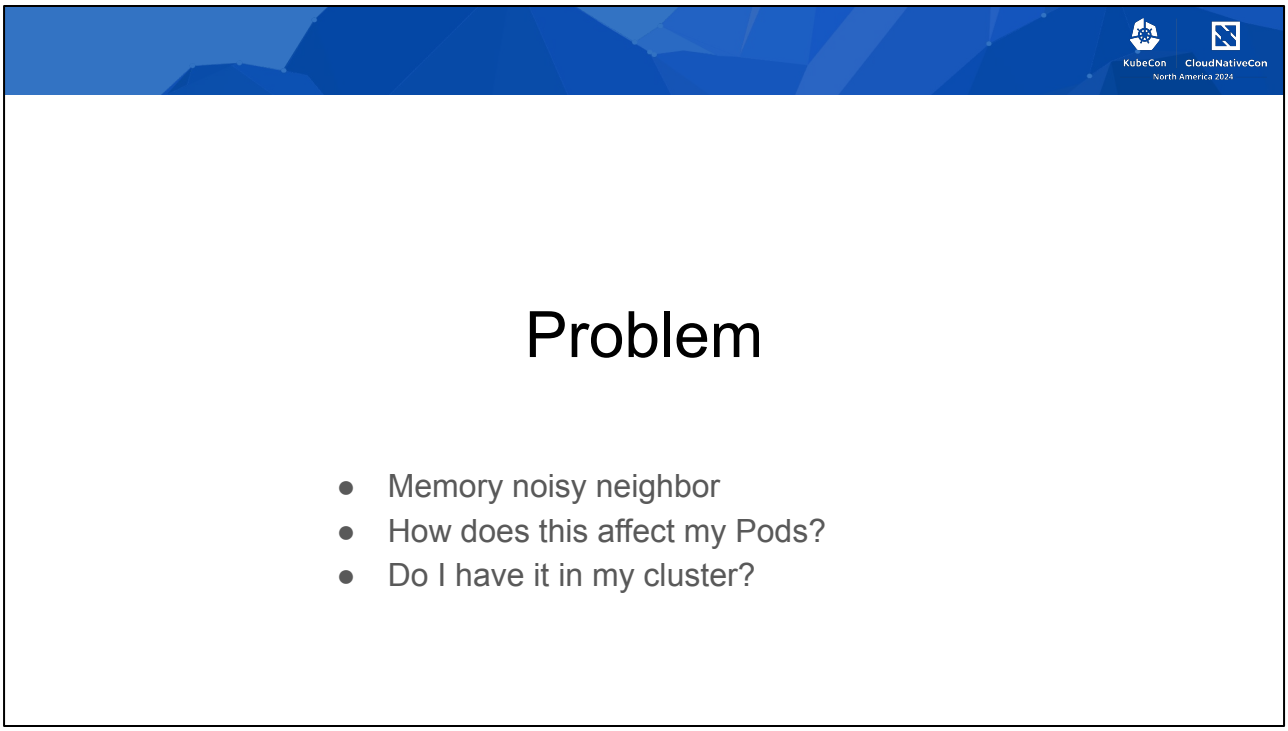


I've divided the talk into **4 sections**, and instead of diving deep into one system, one implementation and giving you all of the details, I've chosen to give a *broad overview* of what's known about this problem and how to solve it.

[pause]

I hope that you'll leave, not with a very detailed idea of exactly how to solve the problem, but understanding that:

- It's **important**
- And knowing the **tools** that we have



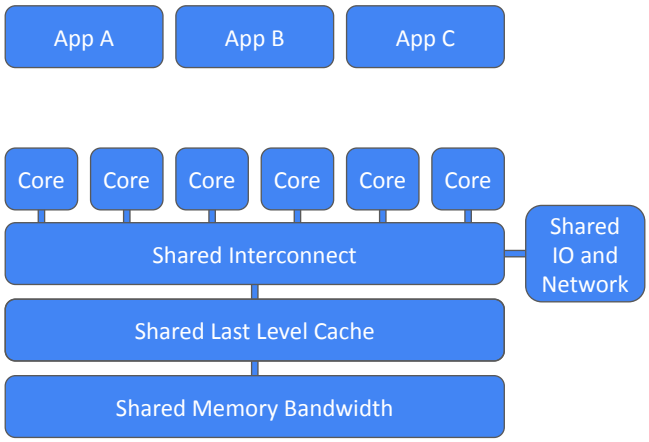
Let's start with the problem:

- What is noisy neighbor?
- What is the effect that it has on pods?
- And do I actually have it in my cluster?

Noisy neighbor: tragedy of the commons

Apps access physical resources

Shared resources are constrained



Our cloud-native applications **ultimately** run on physical hardware.

These applications share **finite hardware resources** that servers have.

Noisy neighbor: tragedy of the commons

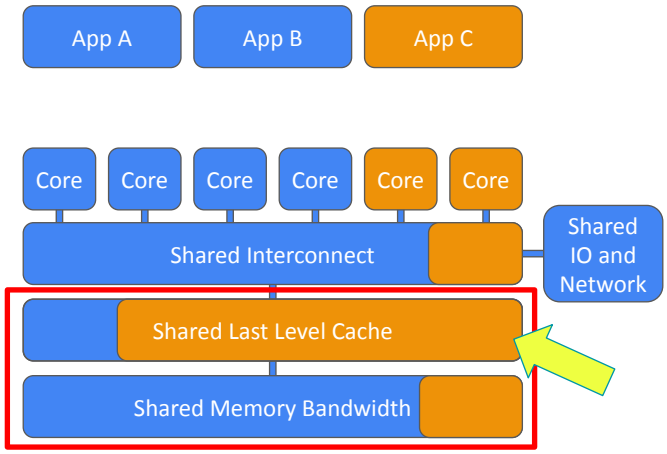
Apps access physical resources

Shared resources are constrained

One App can use more than its fair share, degrading others

This talk:

- Cache
- Memory bandwidth



In noisy neighbor, **one application consumes a lot more than its fair share**. And so the other applications running alongside it:

- Are unable to get the resources that they need
- Have degraded performance

And in this talk I'm going to focus on the **memory subsystem**, which is:

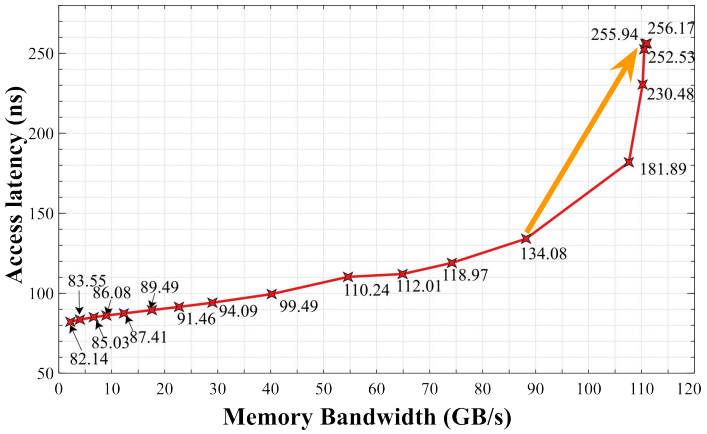
- The Last Level Cache, which I'll just call cache
- And memory bandwidth

Latency increases with memory bandwidth

Change memory bandwidth,
Measure latency

Knee-point around 80%

80% → 100% bandwidth
latency doubles!



W. Tang et al, “Themis: Fair Memory Subsystem Resource Sharing with Differentiated QoS in Public Clouds,” ICPP ’22. doi: 10.1145/3545008.3545064.

In this experiment from 2022, the benchmark workload varied how many reads and writes it performedchanged on the machine such that it used **more and more memory bandwidth**. It created more reads and writes per second, and the experiment measured what is the access latency to memory.

[pause]

As you can see, as the system becomes more and more busy - as the benchmark application reads and writes more and more memory - **the latency increases**.

And around 80%, the 90 GB per second mark here, you can see that there's a **knee point** where the access latency increases *very substantially* with an increase in memory bandwidth.

[pause]

What this means is, if you have one application on the system that is a noisy neighbor and it drives the bandwidth from this 80% mark to the 100% mark, it can **double the memory access latency** for everybody on the system. And this is essentially memory bandwidth noisy neighbor - one application causing high access latency for all others.

CPUs try to protect us from memory latency:

- Prefetchers
- Reorder buffer (ROB)
- Caches

Are we protected? **…or not?**

But you know, CPUs have protection mechanisms for memory latency:

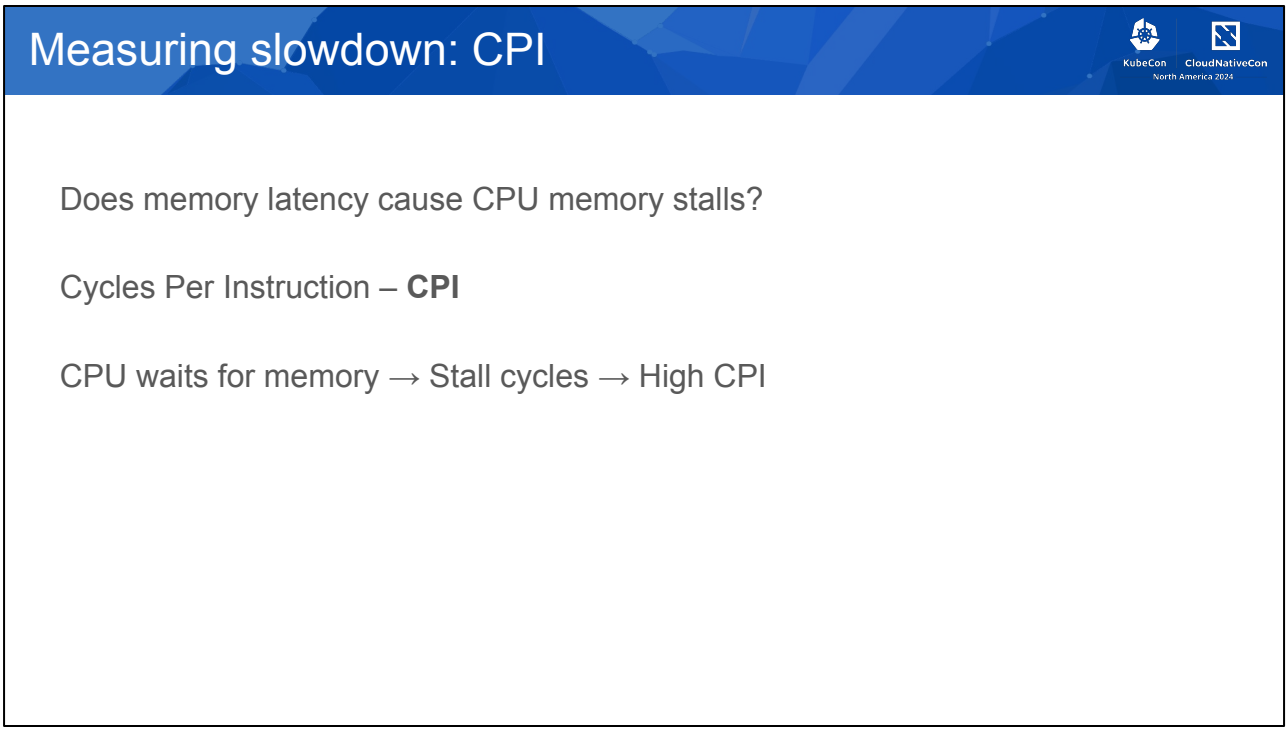
- You have **prefetchers** that try to get the memory from DRAM onto the CPU, so that it's available just as you need it
- There's the **reorder buffer** that allows the CPU to execute instructions out of order while it's waiting for memory, so it can do useful work
- And there are **caches**

[pause]

And the question is:

- Do these mechanisms actually protect us or not?
- Does our performance degrade because of high memory latency?

Measuring slowdown: CPI



There's a very popular metric to measure whether CPUs degrade due to memory, which is **cycles per instruction**, or CPI.

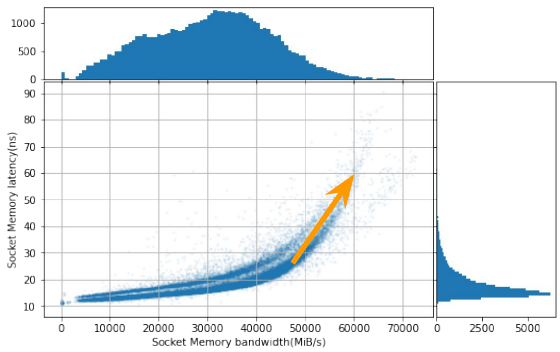
The idea is that as CPUs wait for memory:

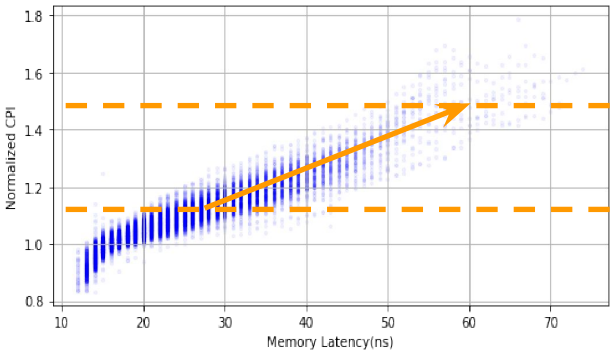
- They cannot do useful work
- So they enter these stall cycles
- So the cycle count increases, while the CPU is not retiring instructions

[pause]

So when there is high memory contention, the **ratio of cycles to instructions becomes higher**.

80% bandwidth cap → 25% more compute-efficient





Memory bandwidth only – has cache mitigation

K. Wang *et al.*, "Characterizing Job Microarchitectural Profiles at Scale: Dataset and Analysis,", *ICPP'22*. doi: 10.1145/3545008.3545026.

Here are measurements taken by Alibaba. This is a trace that Alibaba published in 2022, where they measured:

- A cluster with over **8,000 physical hosts**
- Running over a **million containers**
- And they sampled low-level metrics like:
  - Memory bandwidth
  - Memory latency
  - Cycles per instruction
- On **every context switch** over 24 hours

[pause]

What you can see here on the left is a graph showing:

- On the X-axis, the memory bandwidth
- And the Y-axis memory latency

And this is similar to the graph we saw 2 slides ago. You can see:

- Latency increases as memory bandwidth increases
- And then there's the knee point, and latency increases more dramatically

[pause]

On the right hand side is this graph plotting:

actually increases as well.

And if you look at what noisy neighbor means in this environment:

- If you have a noisy neighbor that drives your bandwidth this last 20%
- It increases latency to around **2x**
- And this causes the cycles per instruction to increase more than **25%**

[pause]

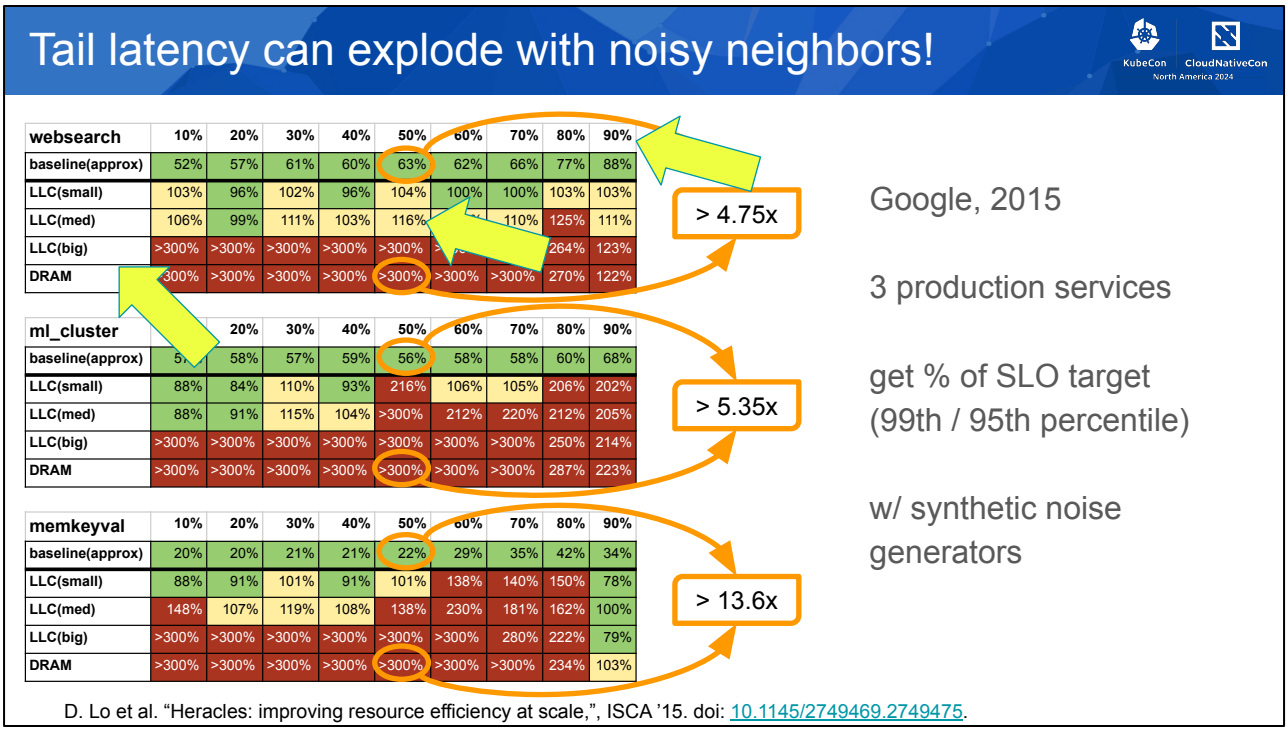
So this means just because you have this one noisy neighbor:

- All the other applications are going to consume more cycles to get their memory
- Around **25% more cycles**
- So *everything becomes 25% slower*

[pause]

And I should say - the system measured by Alibaba Cloud here **already has cache noisy neighbor mitigation**. So this only measures the effect of memory bandwidth noisy neighbor.

If this system didn't have cache noisy neighbor mitigation, like all of our systems don't, then your improvement is going to be a lot more than 25%.



So does this also affect p99 latency, or just average latency? Well, it turns out that the effects on p99 latency are **very, very substantial**.

This is measurements published by Google in 2015 in this paper called Heracles.

[pause]

The 3 tables that you see here correspond to **3 production workloads** at Google:

- The **Search Node** for web search
- **ML Cluster** is a machine learning task classification system that works in real time, so it answers users' requests
- And **MemKeyVal** is an in-memory key value store like Memcache

[pause]

The lines in these tables are the different noise generators that they use:

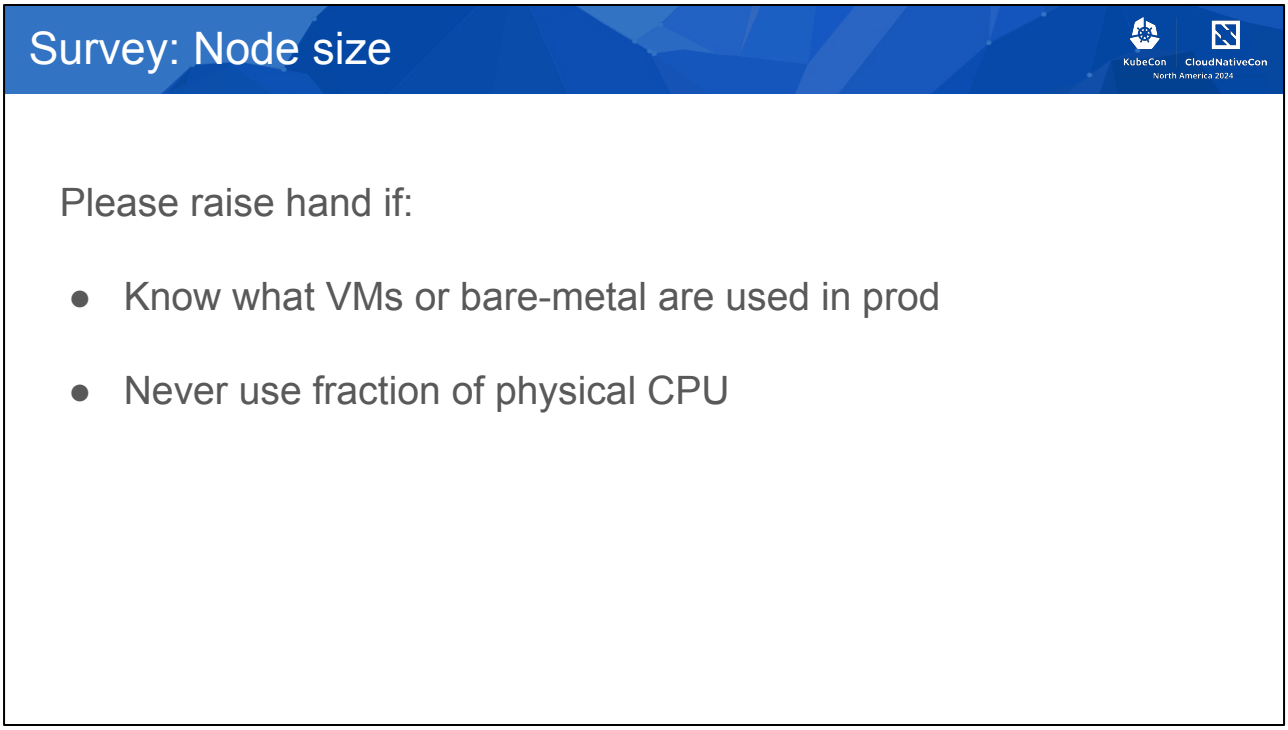
- Baseline without a noise generator
- Then 3 cache noisy neighbor generators: small, medium, and big
- And a memory bandwidth noisy neighbor generator

The columns are load on the system, and the numbers in the table are latency as fraction of their target SLOs. So instead of publishing milliseconds, they publish the percent of what they're aiming for.

[pause]

- ML Cluster more than **5 times**
- And the Memcache equivalent more than **13 times**

This is a *huge increase* in p99 latency because of noisy neighbor.



Let's take another survey. It's the second and last survey for today.

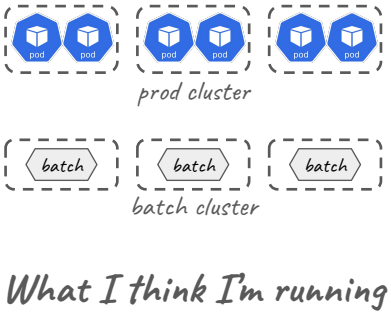
[Interactive moment]

Please raise your hand if you know what type of nodes your cluster is using:

- Are they bare metal?
- Are they the 4 extra larges, 8 extra larges, and so on
- Just a general concept, you don't need to know the exact ones

Now leave your hands up if you always use a whole CPU in those nodes - you're only using the bare metal, or you're only using one half or one quarter of the number of cores on the machine.

Separating batch clusters



What I think I'm running

Best practices today is to **separate** our:

- Big data analytics
- Our batch workloads From our production clusters with user-facing services

[pause]

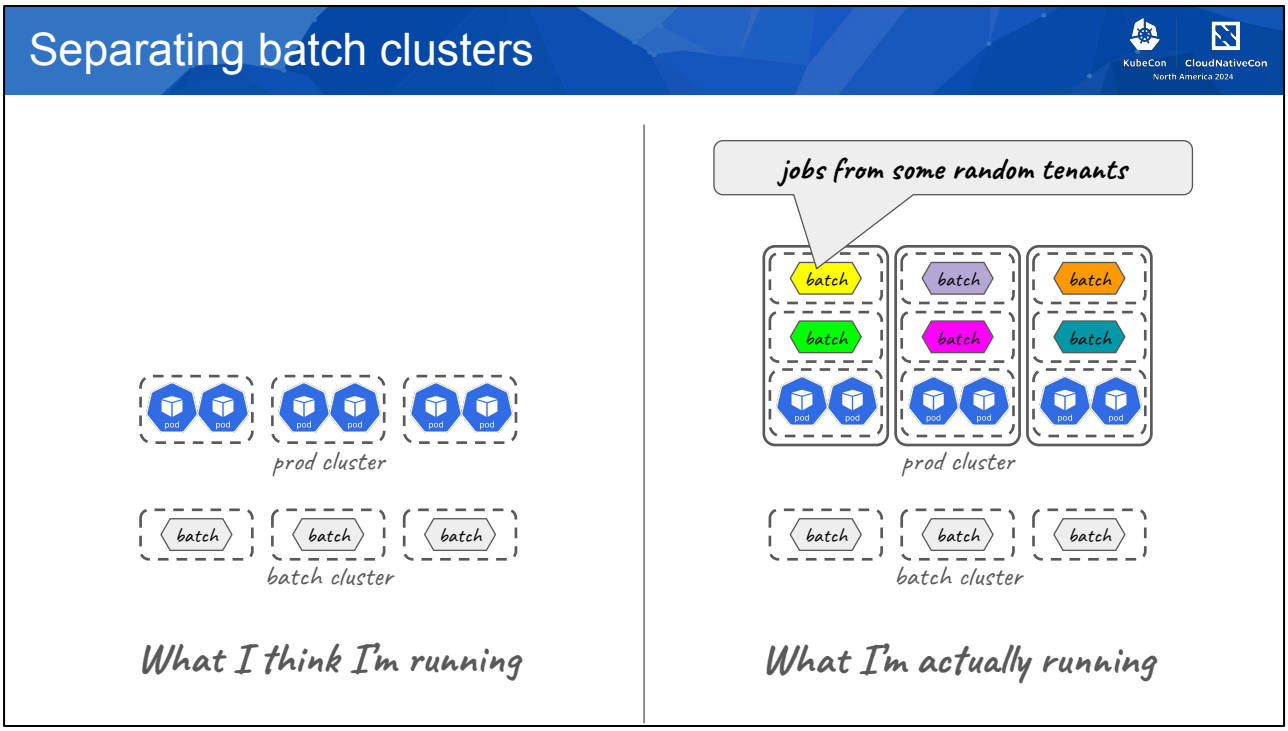
And the reason is you don't want your big data analytics to interrupt your production workload. You don't want somebody to run this big job to destroy performance of your latency-sensitive systems.

[pause]

Well, it turns out, if you're running in this way and:

- You're running on the public cloud
- And you're not taking full CPUs

Then [next slide]



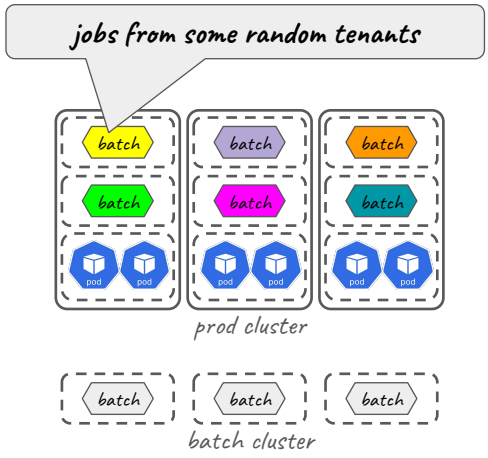
you're running on the same machine with workloads from some random other tenants of your cloud provider or your data center.

[pause]

And so while you've done all the work to separate your batch cluster from your production latency-sensitive cluster, you're actually running with **other people's big data analytics**.

Separating batch clusters

Does your provider protect your VMs from other VMs on the same machine?



What I'm actually running

So it's like this...

[build]

is your cloud provider actually protecting you from memory noisy neighbor caused by other VMs?

I haven't been able to find any mention of this type of protection…

Engineers spend years optimizing user experience

Engineers spend months and years on improving performance for users

They might be:

- Changing, adding database indices
- Changing data schemas in order to make queries faster
- Taking services and splitting them to microservices so processing happens in parallel, and
- Profiling code and alleviating bottlenecks

[pause]

It is absurd that **all this work** can be erased, obliterated by a memory noisy neighbor that reduces performance of the compute infrastructure.

[pause for effect]

Does my cluster have noisy neighbors?

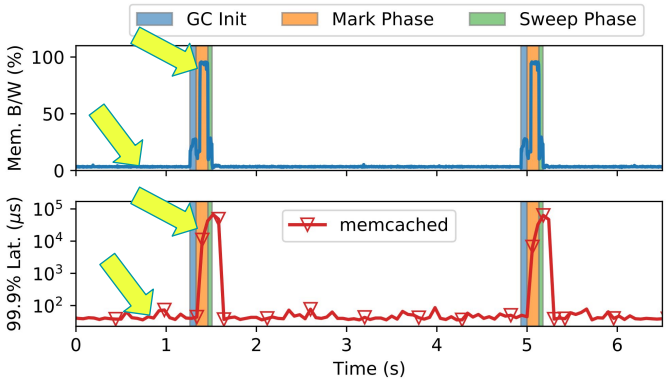
Run:

- Memcached
- garbage-collected workload

Mark Phase is memory-intensive, causes significant slowdown!

Not only “big-data” is noisy

Also: security scanning, video streaming, transcoding…



J. Fried et al., “Caladan: Mitigating Interference at Microsecond Timescales,”. OSDI’20. on usenix.org

Now, do you need to have batch analytics running alongside you in order to experience noisy neighbor?

[pause]

It turns out, no. Even “regular” workloads are noisy neighbors.

In this experiment run by MIT researchers in 2020, they ran Memcache alongside a **garbage collected workload**. You can see on the top graph here the memory bandwidth:

- As the experiment starts, it's pretty low
- And as the mark phase of the garbage collection starts, memory bandwidth is saturated
- This is *very* memory intensive

[pause]

Now look at latency:

- It's usually at 50 microseconds tail latency
- But then, as the mark phase starts, latency spikes **more than 3 orders of magnitude** - *1,000 times*

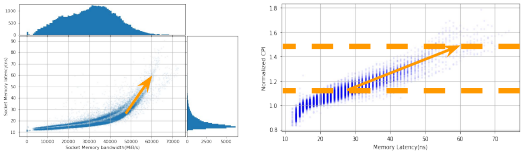
And this shows even a garbage collected workload can be a noisy neighbor to your system.

- Video streaming, transcoding
- Container images come compressed, so when you have to unpack a container - that's memory intensive

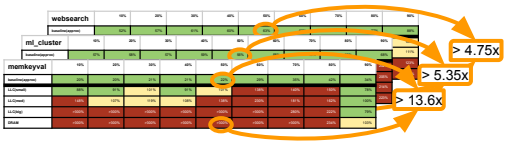
Recap: The Problem

KubeCon CloudNativeCon North America 2024

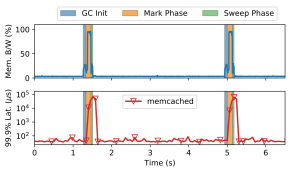
High memory contention
→ High memory latency
→ High CPI (low efficiency)



Tail latency with noisy neighbor increases 4-13x



Many workloads can be noisy (e.g., garbage collection)



And so to summarize so far.

Memory noisy neighbor shows as:

- High memory access latency
- Which translates to high cycles per instructions
- And this just looks like high CPU - it looks like your system is really working hard
- But actually, it's not doing useful work

[pause]

P99 latency also suffers significantly from memory noisy neighbor:

- And we've seen 4 to 13 times increase in tail latency

[pause]

And you might have noisy neighbor from:

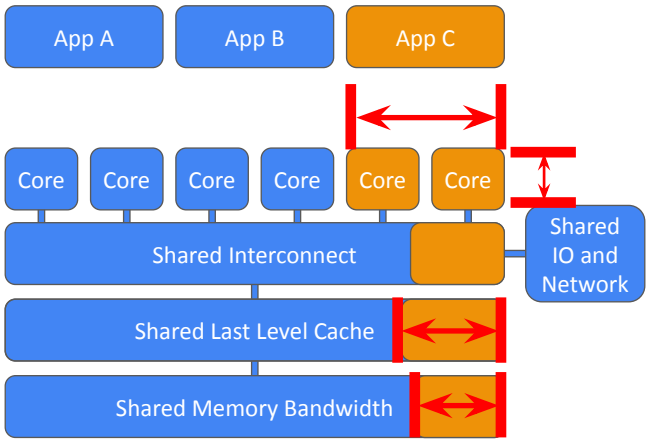
- Other tenants' VMs running alongside you on the cloud
- But also your own workloads can be noisy neighbors

Current support

- Hardware and software mitigation
- Support in Linux

So let's see what's currently available to mitigate noisy neighbor.

Memory system mitigation knobs



Reduce demand (#cycles):

- Core pinning
- Frequency scaling

Direct control:

- Cache allocation
- Memory bandwidth limits

Modern CPUs allow **direct control** of the amount of resources that each application can use. You can:

- Decide what fraction of caches an application can use
- Limit how much memory bandwidth it can use

[build]

And the other method to control noisy neighbor is to **reduce the opportunity** that the noisy neighbor has to consume resources. You can:

- Limit the number of cores it's running on
- Or reduce the frequency of those cores it's running on

This reduces the number of cycles that the noisy neighbor gets. So it can create less noise on shared resources.

Containers (not) to the rescue!

Different focus:

- cycles
- memory capacity (size)

Cgroup v2	Responsibility
CPU	Regulates distribution of CPU cycles
Memory	Controls distribution and accounting of memory usage
IO	Manages distribution of IO resources
PID	Limits the number of processes in a cgroup
Cpuset	Assigns specific CPU(s) and memory nodes
Device	Controls access to device files
RDMA	Regulates distribution and accounting of RDMA resources
HugeTLB	Limits the HugeTLB usage per control group
Perf_event	Allows perf events to be filtered by cgroup path
Misc	Provides resource limiting for scalar resources not covered by other controllers

Reading 1 2

I get this question a lot: Shouldn't *containers help us? Won't cgroups regulate caches and memory bandwidth?*

It turns out that there is a CPU and a memory cgroup, but they're focused on different resources:

- The CPU cgroup is focused on accounting cycles
- And the Memory cgroup mostly accounts for the number of bytes that is used by the application

Linux supports allocation

~~cgroup~~ resctrl allocates:

- Memory bandwidth
- Cache space

Evolved:

- Intel (2016)
- AMD (2018)
- ARM (in progress)

Demo:

- Facebook Resource Control Demo by Tejun Heo and collaborators

Linux resctrl CPU support

	Kernel
Intel RDT Intel Resource Director Technology	v4.10, v4.12
AMD QoS AMD Platform Quality of Service	v5.0
ARM MPAM Memory System Resource Partitioning and Monitoring	WIP, x86→generic Review needed

AMD, ARM

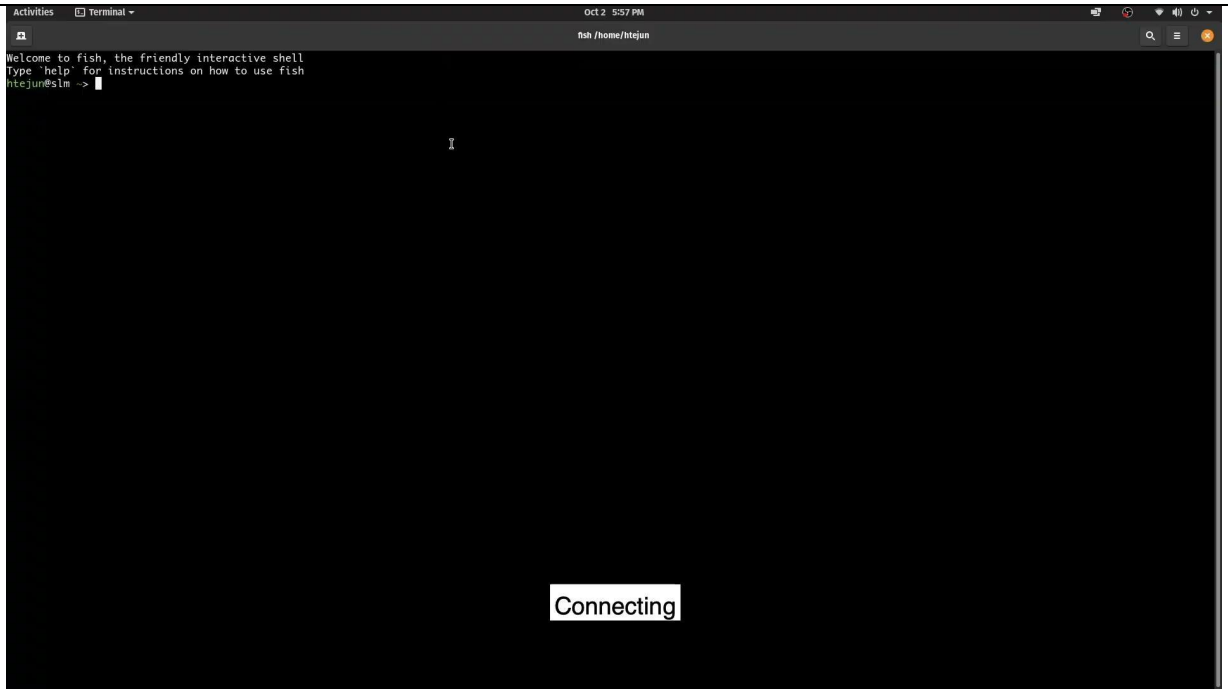
So cgroups does not regulate memory bandwidth and caches.

When Intel wanted to add the support in 2016, kernel maintainers decided to put it in a different subsystem in Linux called **resource control** or **res control**:

- AMD contributed support for their processors in 2018
- ARM is in the process of contributing support for ARM processors

[pause - transition to demo]

I'd like to show you a quick demo. This demo was recorded by folks at Meta, led by Tejun Heo - kudos to those folks for creating this demo. It's called the Facebook Resource Control Demo.



Credit: Meta; Tejun Heo and collaborators. [resctl-demo on GitHub](#).

[Narrating the demo]

Here you can see the system:

- A workload is starting
- On the left you can see in green transactions per second, and in blue latency
- The workload converges to a good point
- And then noisy neighbors start
- The transactions per second falls latency increases

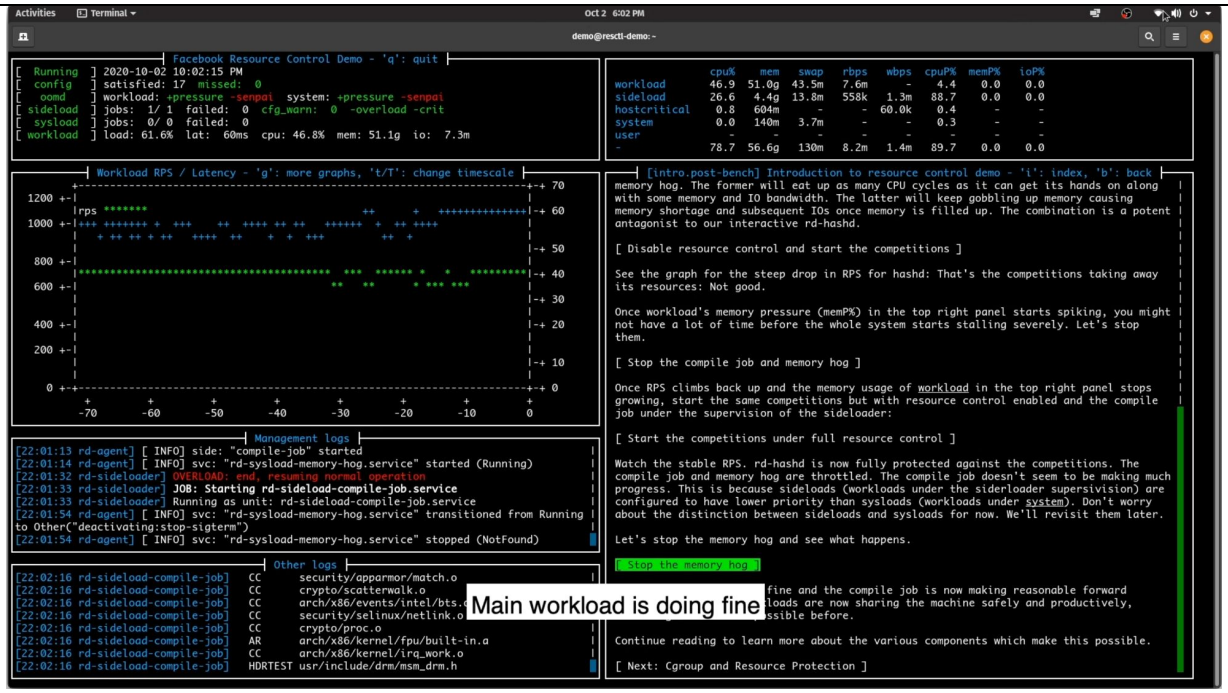
[pause]

Now let's:

- Stop this noisy neighbor workload
- Let our latency sensitive workload converge again
- And now start noisy neighbors with resource control

You can see:

- The workload converges
- It doesn't get interference
- You can see here the compilation job, the memory intensive job
- And this system behaved well



Credit: Meta; Tejun Heo and collaborators. [resctl-demo on GitHub](#).

Hypervisors support allocation

- VMware in Telco Cloud Automation

  → Telco + Finance?
- Static

Intel RDT support in hypervisors

	Cache partitioning	Memory bandwidth
Xen	✔	✔
KVM	✔	
VMware	✔	Monitor (vSphere)
Hyper-V		Monitor PMU
ACRN	✔	✔

Telco Cloud Automation, KVM libvirt NUMA tuning, Xen memory bandwidth, Xen LLC, ACRN, Hyper-V PMU

Hypervisors have ***some*** support for memory and cache allocation.

But there is such a scarcity of documentation. I believe this feature is being used more for very specific niche use cases like Telco and finance. At least, that's my impression from reading the documentation.

VMware:

- vSphere has memory bandwidth monitoring
- Static LLC in Telco Cloud Automation

KVM:

- Static LLC allocation

Xen:

- Tech Preview of memory bandwidth and LLC allocation

ACRN (small IoT/Edge/embedded hypervisor):

- Memory bandwidth and LLC

Mitigation Systems

- Type 1: cycles per instruction (CPI)
- Type 2: latency control
- Type 3: usage control

Okay, so we saw that:

- There is Linux support
- Hardware has support for direct allocation

What systems can you build that control noisy neighbor?

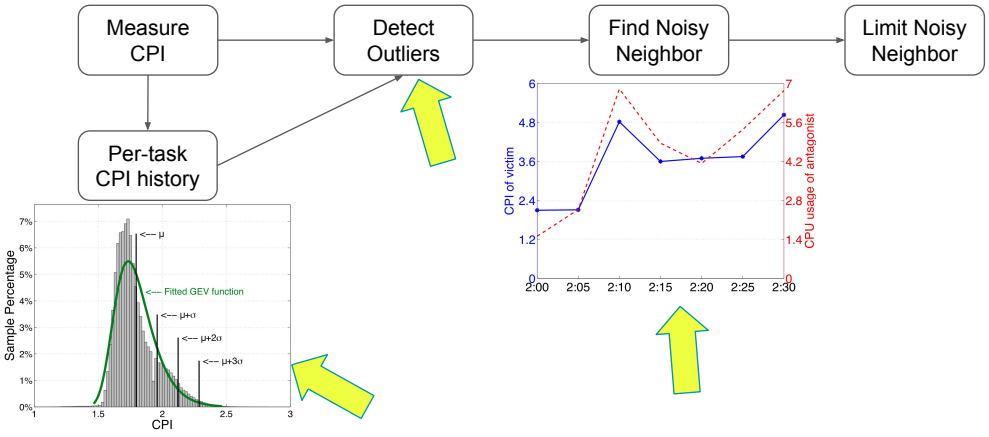
[pause]

I'll go through these **3 major types** of systems. If you read papers, most of the systems will fall into one of these 3 categories.

Type 1: Cycles Per Instruction (CPI)

KubeCon CloudNativeCon North America 2024

Uses: High interference → high CPI



X. Zhang et al., "CPI ^{2}: CPU performance isolation for shared compute clusters," in *EuroSys 2013*. doi: 10.1145/2465351.2465388.

Let's start with the **cycles per instruction based mitigation system**.

The idea here is:

- When there is high memory interference
- There's high cycles per instruction
- This is what we talked about before.

[pause & build]

The major issue that the system has is that it doesn't know what a **good CPI** is and what a **bad CPI** is. So it needs to create baselines.

- The system measures the CPI for every task
- And then creates these profiles of what this usually looks like

[pause & build]

- The system then uses these profiles to detect outliers
- A task is an outlier when it has an abnormal CPI reading for several measurement interval.

[pause & build]

Once the system detected an outlier:

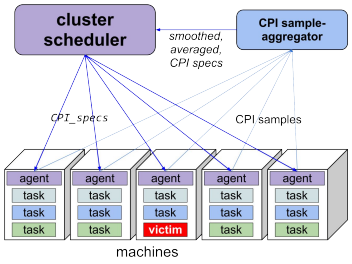
- It tries to find what is the noisy neighbor affecting that outlier

[pause]

The major issue with this system is that CPI and CPU measurements are very noisy. So they need to be averaged over multiple minutes to make good decisions.

But by the time the system finally makes that decision, a lot of damage is done. So that is the major disadvantage of CPI-based systems.

Type 1: Cycles Per Instruction



"We have rolled out CPI^2 to all of Google's shared compute clusters." – paper authors, 2013 @Google

	Type 1: CPI
Measurement	Cycles, Instructions
Averaging	High
Cluster components	Aggregator
Pros	• Simple to measure
Cons	• Complex deployment • Averaging → Slow reaction

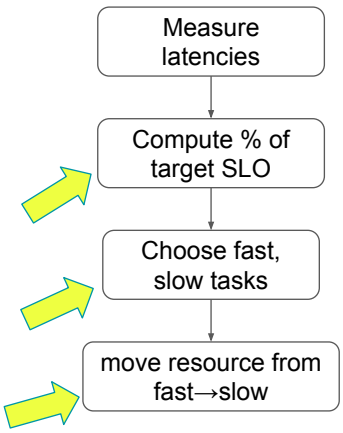
X. Zhang et al., "CPI ^{2}: CPU performance isolation for shared compute clusters," in *EuroSys 2013*. doi: 10.1145/2465351.2465388.

Another disadvantage is that the system needs to create these profiles. So it has centralized components to aggregate measurements. This makes deployment complex.

According to Google, in this 2013 paper, as of paper writing, this had been deployed to all of Google's shared clusters.

Type 2: Latency Control

Example algorithm:



	Type 2: Latency
Measurement	App latency
Averaging	Medium
Cluster components	Node only
Pros	• No profiling → Node-local • Control what you care about
Cons	• High developer effort • Noisy signal → Averaging

S. Chen et al. "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in ASPLOS '19. doi: 10.1145/3297858.3304005.

Let's move to the second type of system: **latency control**

This system:

- Measures the application layer latencies, which is what we're trying to optimize
- We want to hit our latency targets

It computes the percent of the target SLO:

- So maybe one application is at 50% of the max 95th percent latency
- And another one is 98% of the target latency

[pause]

And then it:

- Finds the fastest and slowest task as compared to their target SLOs
- And moves resources from the fast to slow

So you can think about it as Robin Hood:

- If you have one app that's super slow and struggling
- And another application that's super fast
- Then it moves resources between them

[build]

But:

- Application layer latency is noisy
- Maybe the application makes calls to other services or databases
- So we still need to average many samples to get a good reading

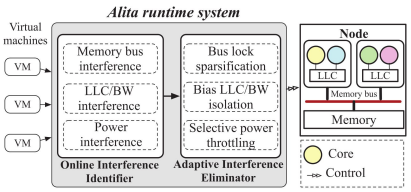
[pause]

And it's usually ***hard*** for an organization to deploy latency measurements across every service in a uniform way.

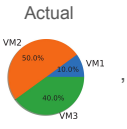
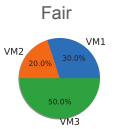
Type 3: Usage Control

How:

- Measure per-app resource usage
- Find unfair allocation
- Limit offender



Cache example:

If similarity ( , ) < 0.8,

then limit cache polluters to their fair share

Q. Chen *et al.*, "Alita: Comprehensive Performance Isolation through Bias Resource Management for Public Clouds," in *SC20:* link.

The **third category** is what I call usage control. The system

- **Explicitly** measures cache usage and memory bandwidth
- Find if there are applications using too much
- And then limit those applications

[pause]

For example, here is an example with cache occupancy:

- You measure what is the actual cache occupancy of your applications
- And then you compute what is a fair allocation
- So maybe proportional to the number of cores that each application is allocated

[pause]

You then apply a similarity function.

- If the actual measurements are close to the fair allocation, then you don't need to do anything
- But if they're far away, if they're not as similar
- Then you limit the abusers to their fair share

Do we really want a fair allocation?

There is a question: *do you really want fairness*? Maybe some applications need some preference versus others.

[pause]

But remember that the goal of the system is to limit these **egregious** behaviors by the super noisy neighbors, and so just limiting every application to around its fair share should be sufficient:

- In order to avoid these worst behaviors
- And move the system away from the high CPI, high tail latency regime.

Type 3: Usage Control (cont'd)

	Type 3: Usage
Measurement	CPU counters
Averaging	Low
Cluster components	Node only
Pros	• Node-local • Measure directly → Fast reaction → Easy to reason
Cons	• Do we really want fair allocation?

Deployed in production, > 2 years:

- 30k nodes (24 to 48 cores each)
- 250k VMs

- authors, 2020 @Alibaba Cloud

Q. Chen *et al.*, "Alita: Comprehensive Performance Isolation through Bias Resource Management for Public Clouds," in *SC20:* link.

And so the benefit here is that these systems are:

- Very easy to build
- They do not require centralized components
- You measure the resources and control those resources directly
- So they're simple to reason about
- And they can react very quickly

[pause]

According to this paper from 2020 in Alibaba Cloud, this has been deployed to around order of magnitude of a million cores in production for over 2 years.

Summary: Mitigation Systems

	Type 1: CPI	Type 2: Latency	Type 3: Usage
Measurement	Cycles, Instructions	App latency	CPU counters
Averaging	High	Medium	Low
Cluster components	Aggregator	Node only	Node only
Pros	• Simple to measure	• No profiling → Node-local • Control what you care about	• Node-local • Measure directly → Fast, Easy to reason
Cons	• Complex deployment • Averaging → Slow reaction	• High developer effort • Noisy signal → Averaging	• Do we really want fair allocation?

- The three “categories” of published systems
- Many good ideas → general purpose
- Usage control (#3) is promising

So to summarize, we have these **3 categories** of mitigation systems.

And if you read papers:

- There are a lot of good ideas
- There are enough good ideas out there that we can cobble them together to build a good solution for the Kubernetes ecosystem

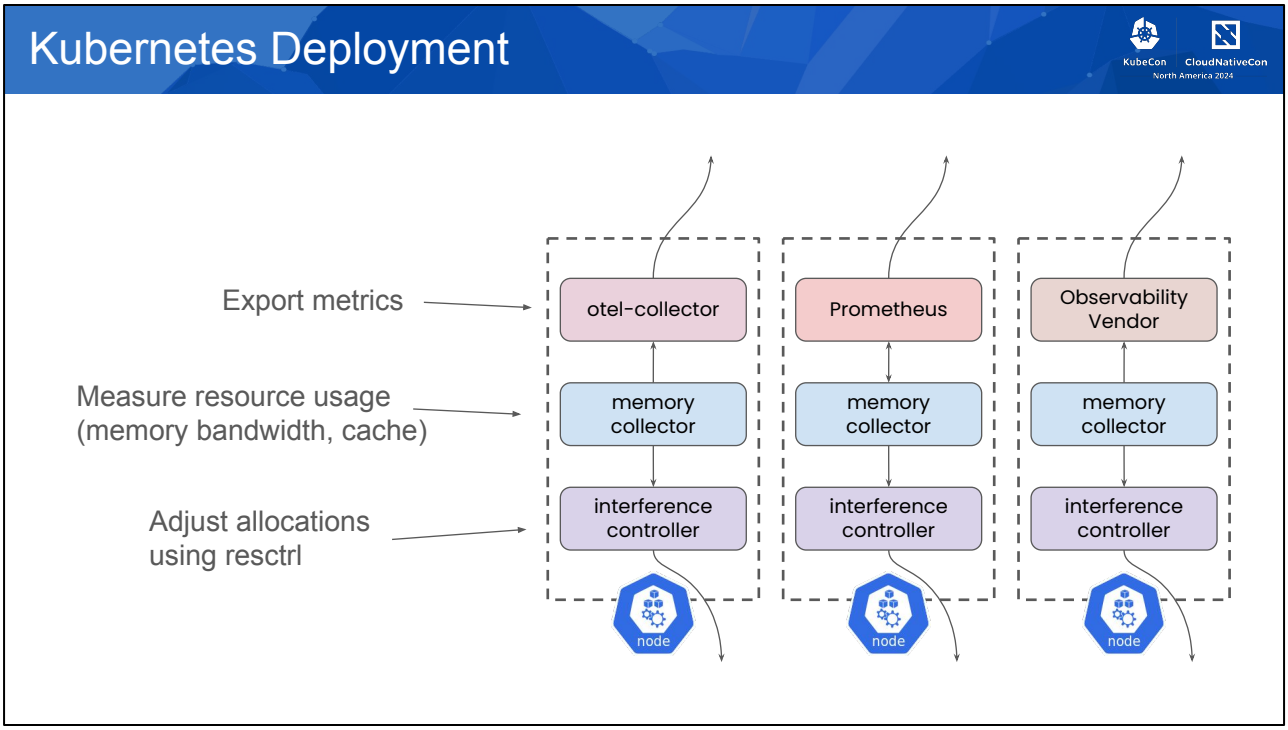
[pause]

And especially, I like this **third type** of category, it is simple and effective. So I think as a community, we should go towards type 3 systems.

Next Steps

- Kubernetes Deployment
- Observability
- Community Next Steps

Okay, so let's talk about what are next steps - what we should do in the community in order to get the benefit from noisy neighbor mitigation.



How would a Kubernetes deployment look?

You'd have - and this is repeating across most of the systems that I've seen:

- One component that I've called the **memory collector** here that measures the resource utilization
  - For example, memory bandwidth
  - And the cache actual utilization per application
- There's then another component which I call the **interference controller**
  - Which adjusts allocations using resource control
  - The Linux resource control subsystem

[pause]

Now, this system should also have another component. And that's **observability**:

- Where you take the memory metrics
- And you export them to different backends

[pause & build]

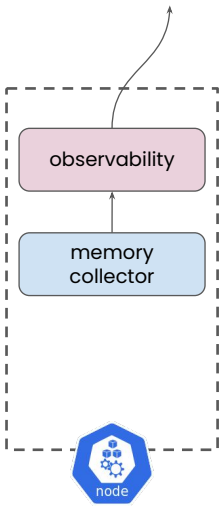
And in fact, I think we should **start with observability**.

First step

Start with observability, because:

To deploy with confidence, need

- Quantify the benefit
- Metrics to show it is behaving correctly



I think we should start with observability because

in practice, to get adoption, operators will want to:

- First, know how much benefit they will get from adding a controller to their cluster, and
- Have visibility, when there is an issue in the cluster, that it is behaving as expected

Community Next Steps

KubeCon CloudNativeCon North America 2024

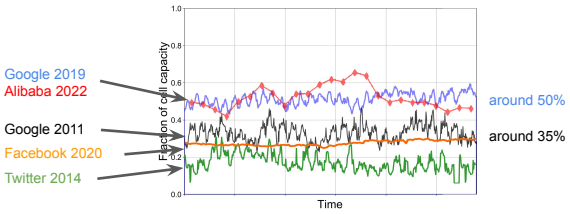
perfpod/memory-collector

Call for:

- Contributors
  - Collector
  - Kubernetes benchmarks
- Potential users
  - Want to hear more
  - Deploy in test/staging

Contact: yonch@yonch.com

Set up time: yonch.com/collector



I started a Github Repo for the memory collector and would like to invite everybody who is interested to participate.

If you want to make repo contributions:

- We are looking for contributors to the collector
- And we'd also like to develop a set of Kubernetes test beds and benchmarks
  - Where we can test the correctness of the collector
  - And show memory noisy behavior as a demo

[pause]

And even if you don't have time to contribute to the repo:

- If you just want to hear more
- Or if you think you can deploy early versions in test or staging environments
  Please reach out. We'd like to make sure that the collector supports as broad a range of Kubernetes environments as possible.

[pause]

So to finish, I want to go back to the hyperscaler CPU utilization graph.

I hope we can drive the community to:

- Better efficiency


KubeCon
CloudNativeCon
North America 2024