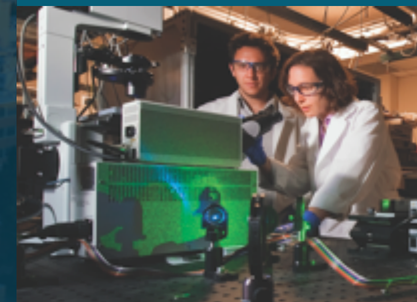
An Initial Examination of the Effect of Container Resource Constraints on Application Perturbation



PRESENTED BY

Scott Levy and Kurt B. Ferreira

SAND2021-6125 C

Center for Computing Research

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

The Promise of Containers

- Containers have become an increasingly popular method for packaging and deploying applications in many environments from desktops to datacenters, clusters and supercomputers.
  - lightweight isolation (relative to VMs)
  - run anywhere (desktop-to-supercomputer)
  - consistent software environment (bring-your-own)
- Docker is probably the best known container runtime. Dockerfiles are the standard way to define containers.
- However, Docker requires a daemon and elevated privileges. Both requirements make it difficult/impossible to deploy on supercomputers and other large shared resources. As a result, several other approaches have emerged for HPC that address these issues.

Charliecloud podman

The Promise of Containers (cont'd)

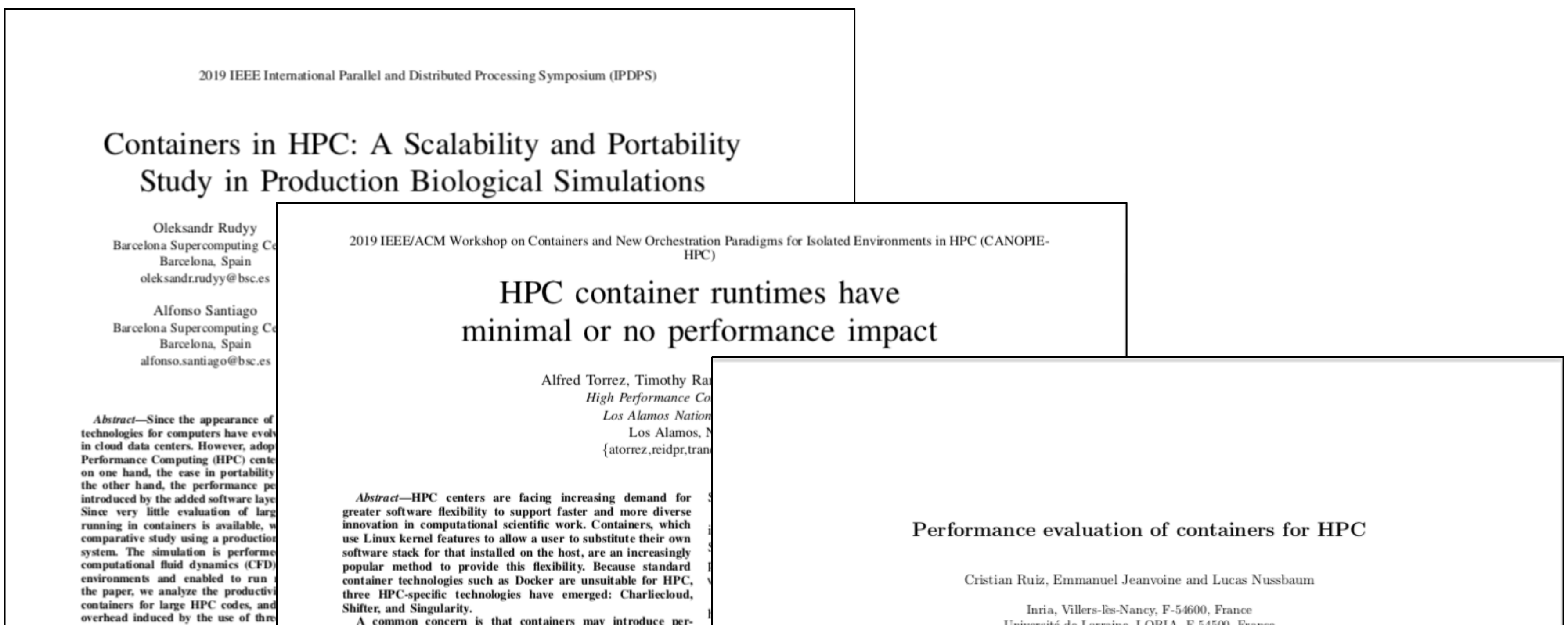
- In addition to the isolation provided by containerization, container runtimes can also be used to partition resources between multiple containers sharing resources.

- For example, in the context of scientific simulations running on HPC systems, resources could be split between a container running the simulation and a second simulation running in situ visualization and analysis applications

- Currently, the most common way that container runtimes partition resources between containers is by leveraging Linux control groups (`cgroups`)

The Performance of Containers

- Existing research provides empirical evidence that suggests that the performance overhead of containerization is modest or nonexistent
- However, most of the data in these papers were collected at relatively modest scales (i.e., a very small fraction of a leadership-class system)
- One significant challenge to running applications on extreme-scale systems is application perturbation (e.g., OS noise : *Ferreira et al.* SC08, *Hoefler et al.* SC10)



Application Perturbation (i.e., Noise) and Performance

- Noise can manifest in different ways (e.g., network, memory) but for the purposes of this presentation, we're limiting the definition of "**noise**" to mean periods of time when a process is deprived of the CPU.

- Existing research has shown that the **duration** of noise has a much greater affect on application performance than its **frequency**. Therefore, all of these results focus on the tail of the noise duration distribution.

- In general, noise events start to have a significant impact on application performance when their duration **exceeds 1 ms**.

Experimental Environment

- We ran experiments with three different container runtimes:

- We ran our experiments on three systems at Sandia : **Stria**, **Eclipse**, and **Mungbean**.
- **Stria** is a development system for Astra (the first petascale Arm system). It has two sockets, each populated with a Cavium Thunder-X2 Arm processor, and a Mellanox ConnectX-5 Infiniband NIC.
- **Eclipse** is CTS-1 system. It has two sockets, each populated with an Intel Broadwell processor, and an Intel Omni-Path NIC.
- **Mungbean** is a Linux workstation. It has a single Intel Sandy Bridge processor and a gigabit Ethernet NIC.

Experimental Environment (cont'd)

- To measure application perturbation we built containers for each runtime that contain **narcissistic**, a Sandia implementation of a selfish benchmark.

- Selfish benchmarks run very tight (and short) compute loops and look for run-to-run variation.

- For each experiment, we run multiple containers concurrently on the same node and record the noise events in each over 15 minutes.

Container Use Cases

WITHOUT RESOURCE PARTITIONING OR CONTENTION

- Ran experiments on Stria and Eclipse with rootless containers with all three container runtimes

WITH RESOURCE PARTITIONING

- On Stria, the `podman` installation uses `cgroups v1` and `runc`. However, root access is required to use `cgroups v1` to partition resources with control groups.
- Using `cgroups v2` to run rootless containers (on a standalone Linux machine) but was unable to figure out how to get the resource limits to actually take effect
- So…we ran our partitioning experiments on a standalone Linux workstation as root using `cgroups v1`.
- The only resource we partitioned was the CPU
- We considered three different mechanisms for allocating CPU resources in podman
  - `--cpu-quota` & `--cpu-period`
  - `--cpu-shares`
  - `--cpuset-cpus`

Container Use Cases

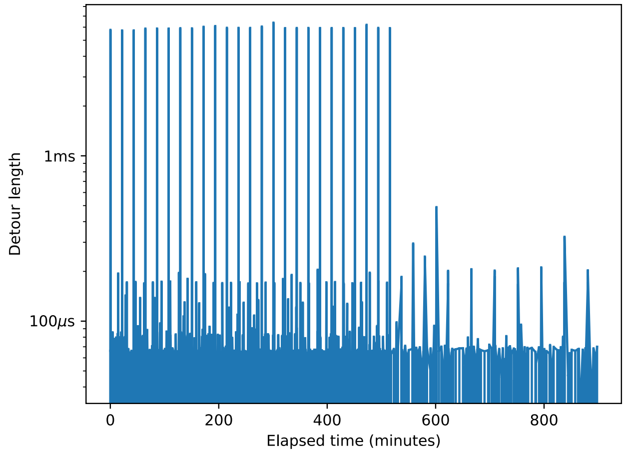
WITHOUT RESOURCE PARTITIONING OR CONTENTION

- Ran experiments on Stria and Eclipse with rootless containers with all three container runtimes

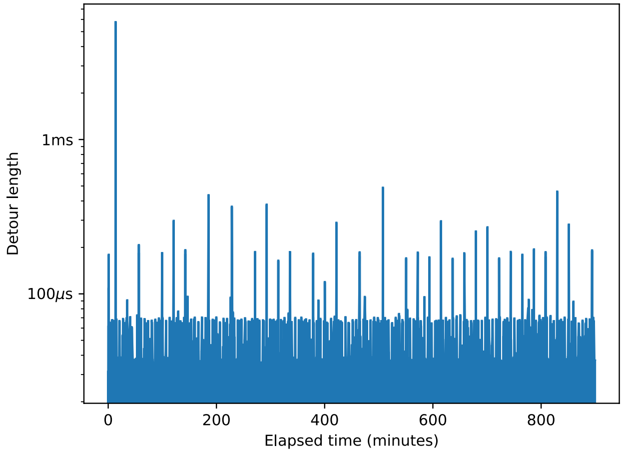
WITH RESOURCE PARTITIONING

- On Stria, the `podman` installation uses `cgroups v1` and `runc`. However, root access is required to use `cgroups v1` to partition resources with control groups.
- Using `cgroups v2` to run rootless containers (on a standalone Linux machine) but was unable to figure out how to get the resource limits to actually take effect
- So…we ran our partitioning experiments on a standalone Linux workstation as root using `cgroups v1`.
- The only resource we partitioned was the CPU
- We considered three different mechanisms for allocating CPU resources in podman
  - `--cpu-quota` & `--cpu-period`
  - `--cpu-shares`
  - `--cpuset-cpus`

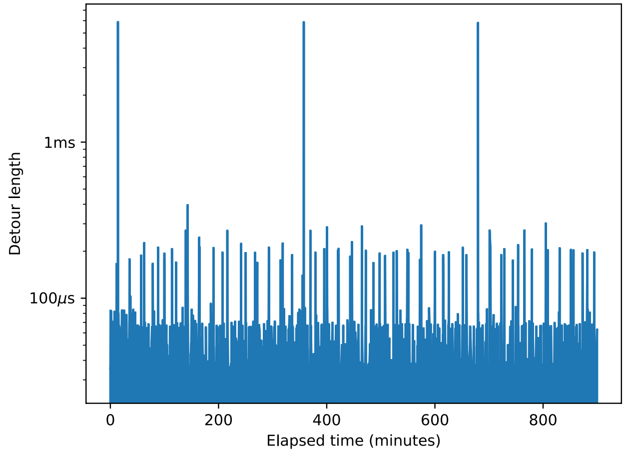
Containers Without Resource Partitioning or Contention (Stria)



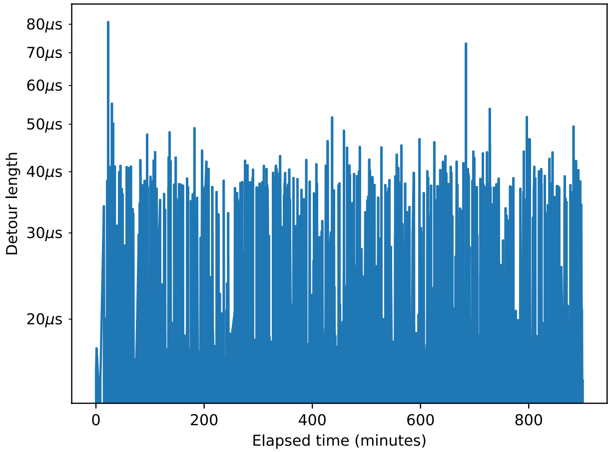
Baseline (w/o container)



Singularity

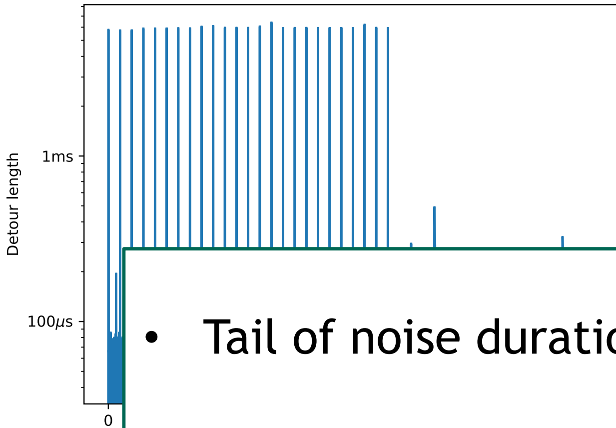


Charliecloud

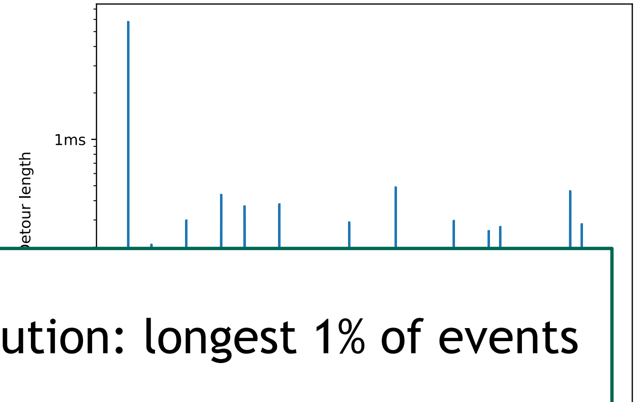


Podman

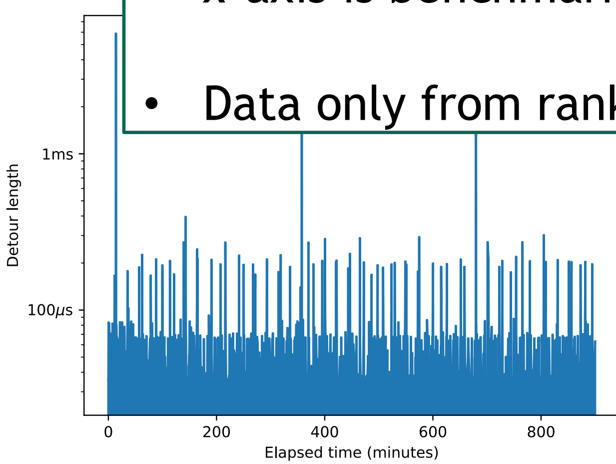
Containers Without Resource Partitioning or Contention (Stria)



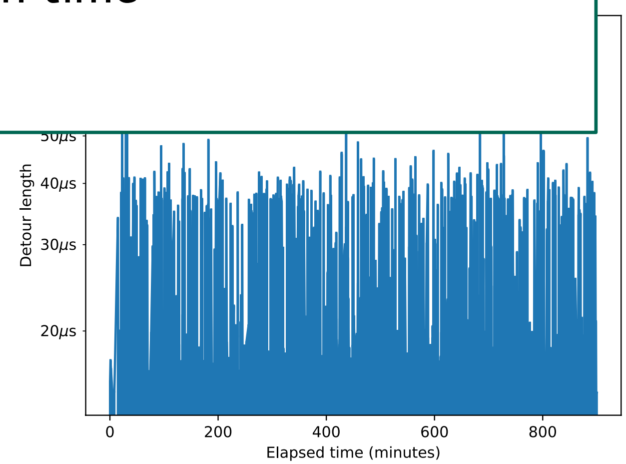
B



- Tail of noise duration distribution: longest 1% of events
- *y*-axis is duration of noise event ($\log_{10}$-scale)
- *x*-axis is benchmark execution time
- Data only from rank 0

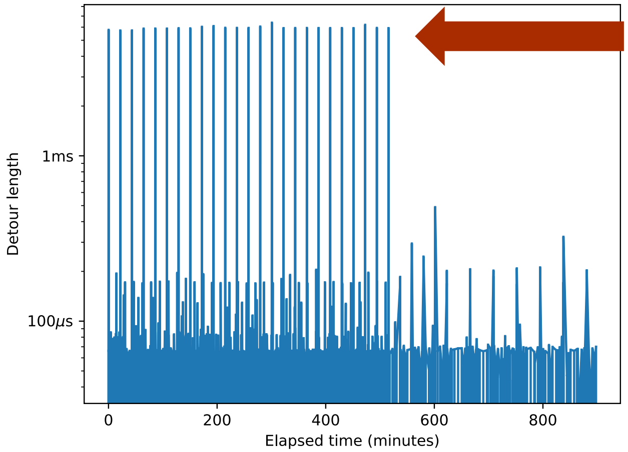


Charliecloud

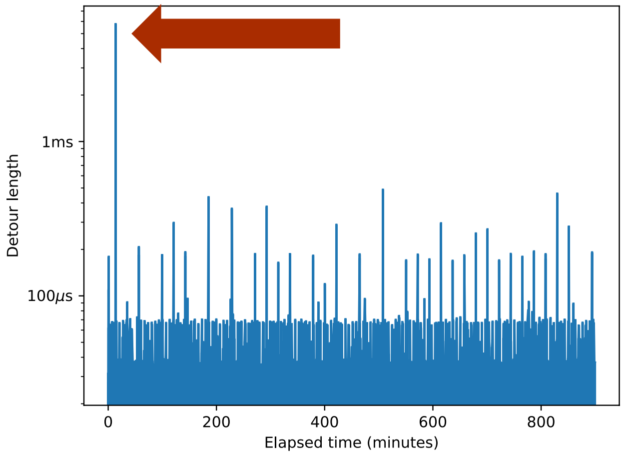


Podman

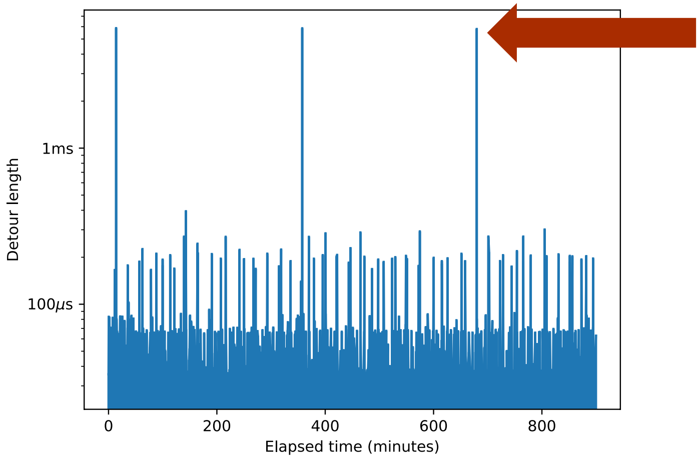
Containers Without Resource Partitioning or Contention (Stria)



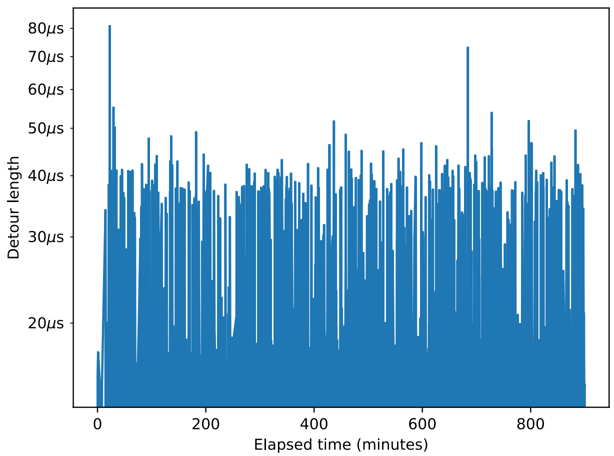
Baseline (w/o container)



Singularity

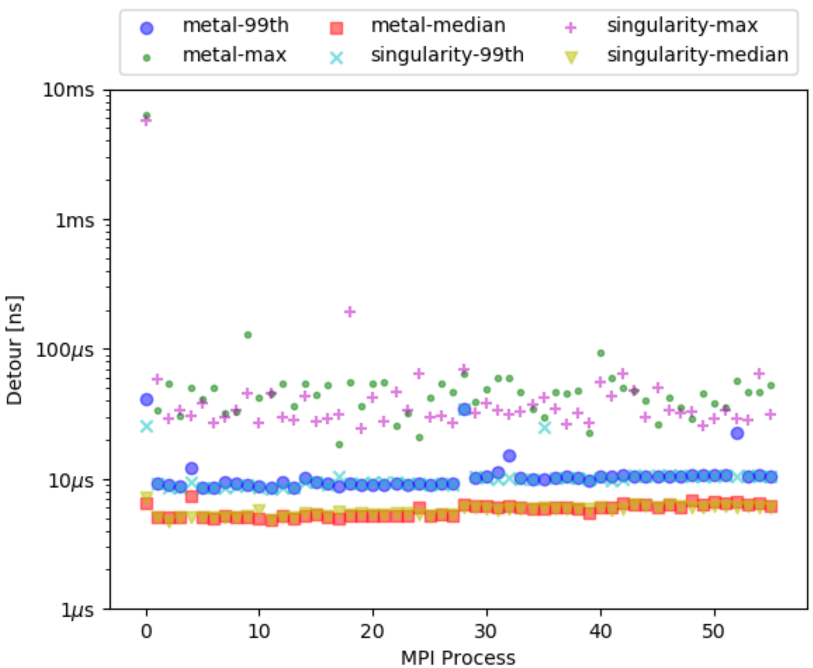


Charliecloud

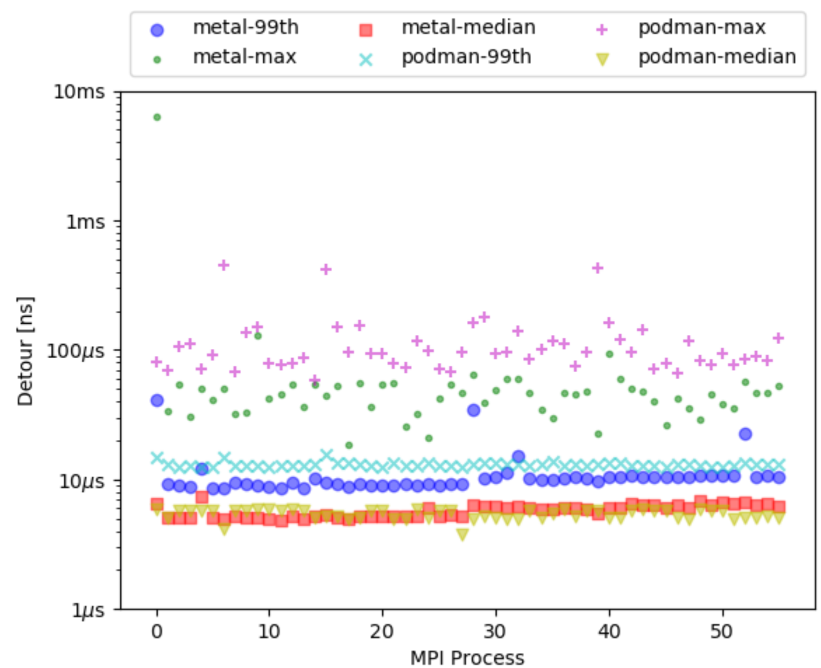


Podman

Containers Without Resource Partitioning or Contention (cont'd)

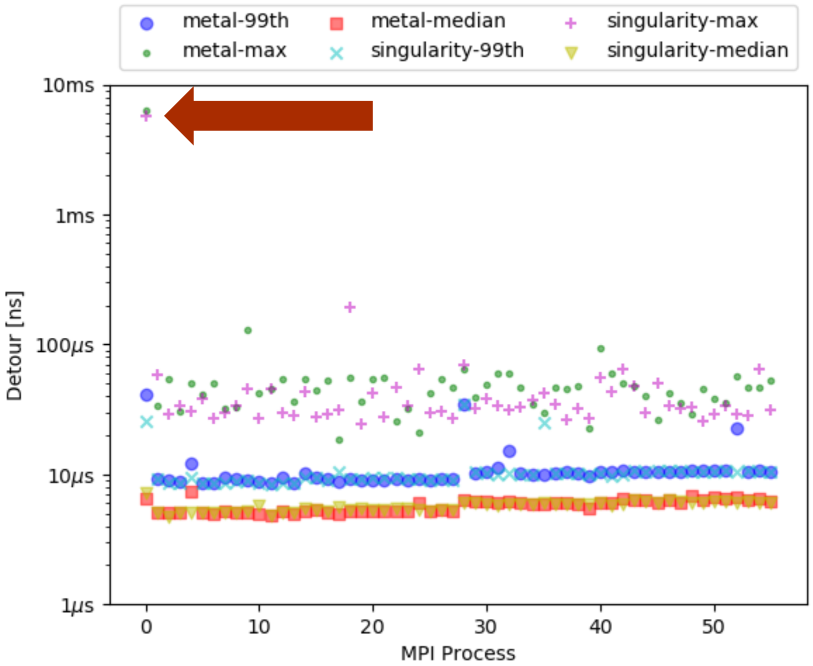


Stria + Singularity

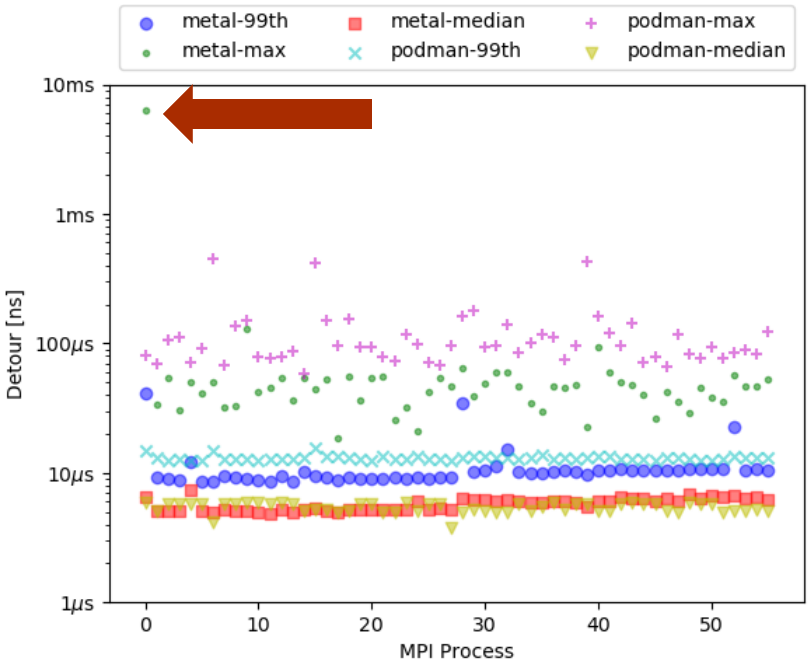


Stria + Podman

Containers Without Resource Partitioning or Contention (cont'd)

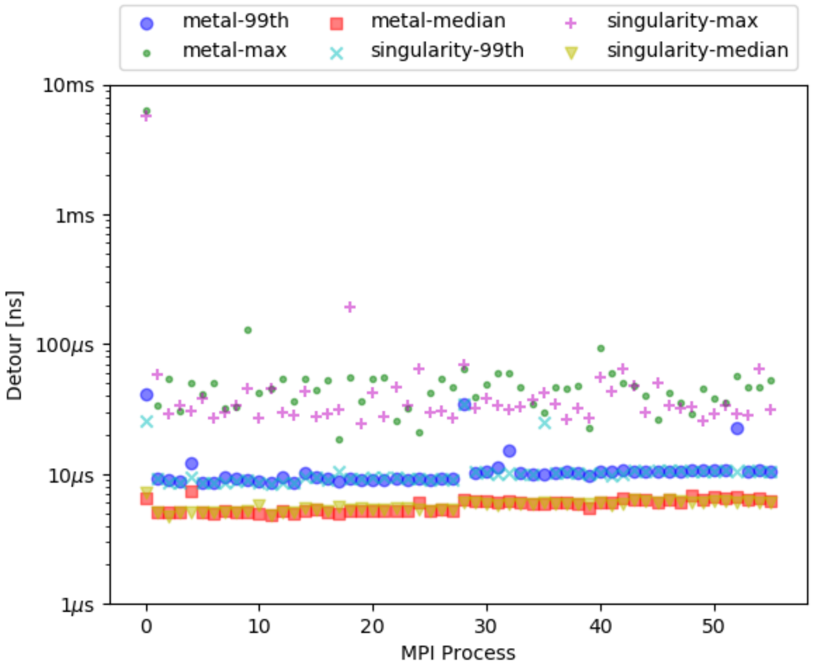


Stria + Singularity

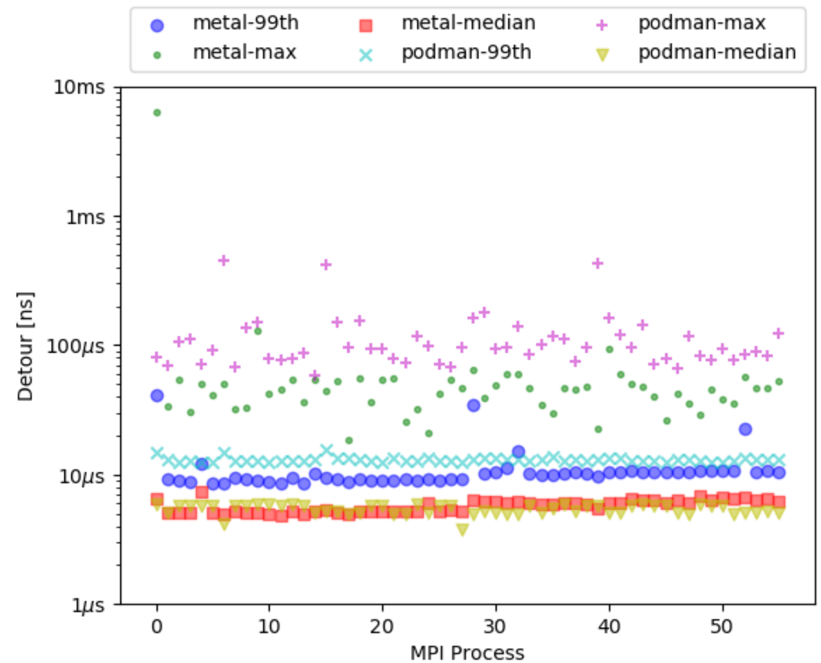


Stria + Podman

Containers Without Resource Partitioning or Contention (cont'd)



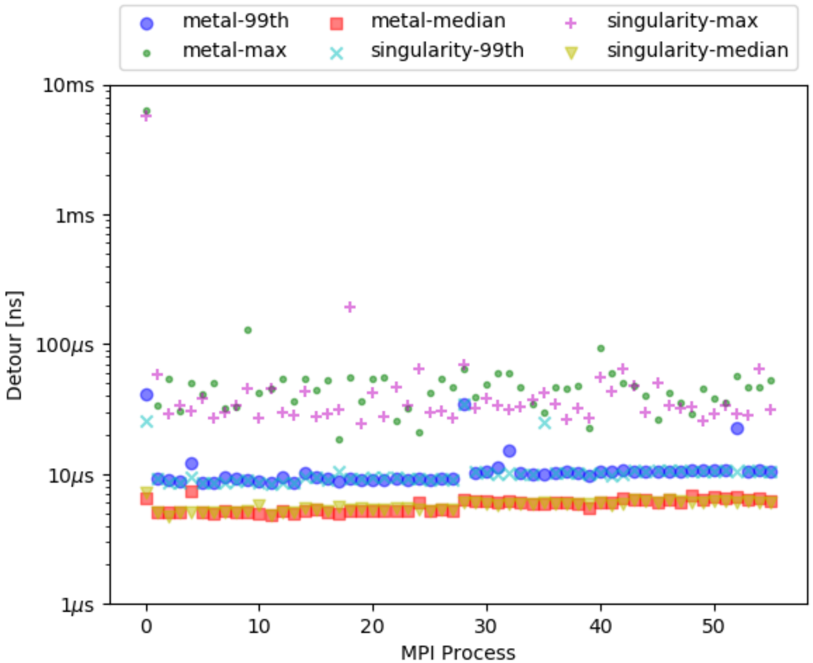
Stria + Singularity



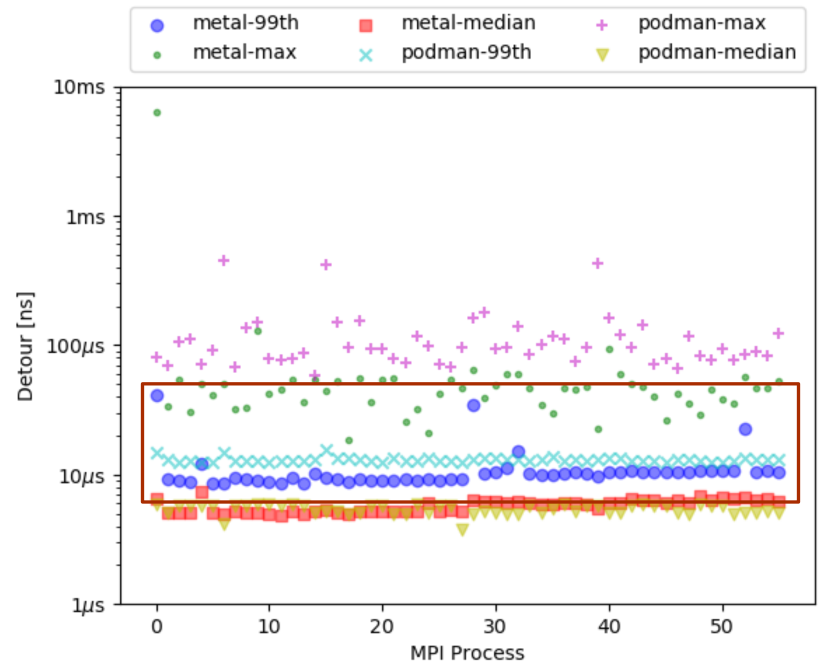
Stria + Podman

Duration distributions are virtually the same for Singularity and the baseline. Where differences exist, the baseline's distribution is slightly more heavy-tailed.

Containers Without Resource Partitioning or Contention (cont'd)



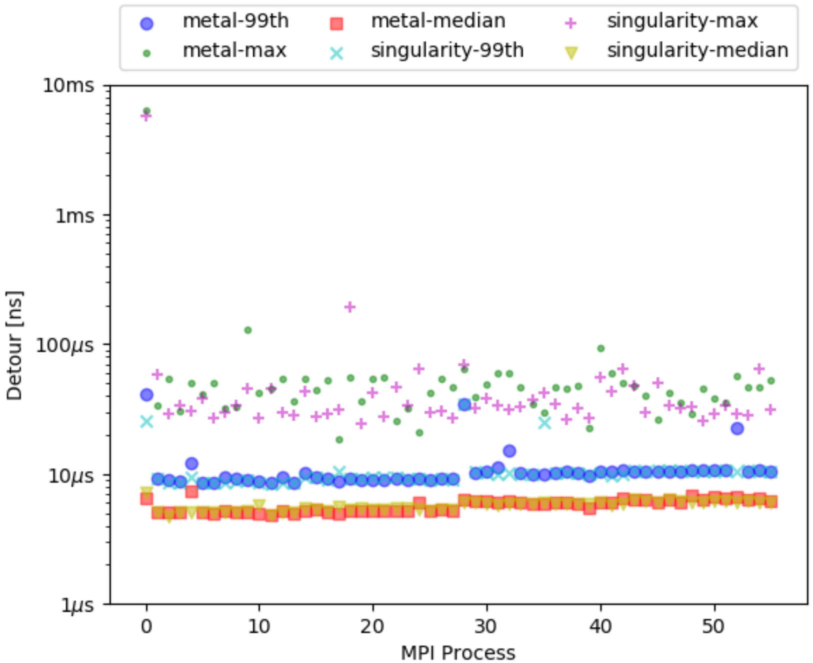
Stria + Singularity



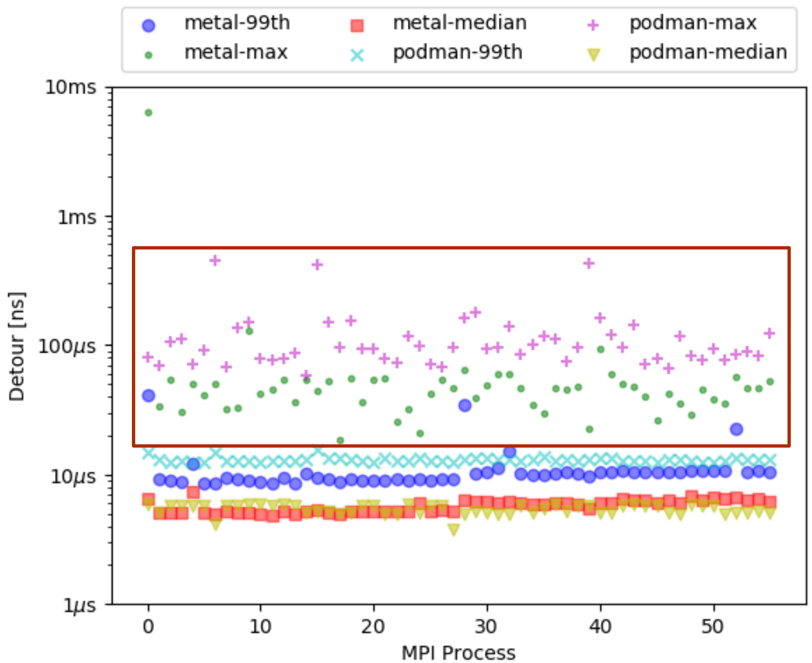
Stria + Podman

Podman's 99th percentile is generally a bit higher than the baseline

Containers Without Resource Partitioning or Contention (cont'd)



Stria + Singularity



Stria + Podman

...same is true for the maximum duration

Container Use Cases

WITHOUT RESOURCE PARTITIONING OR CONTENTION

- Ran experiments on Stria and Eclipse with rootless containers with all three container runtimes

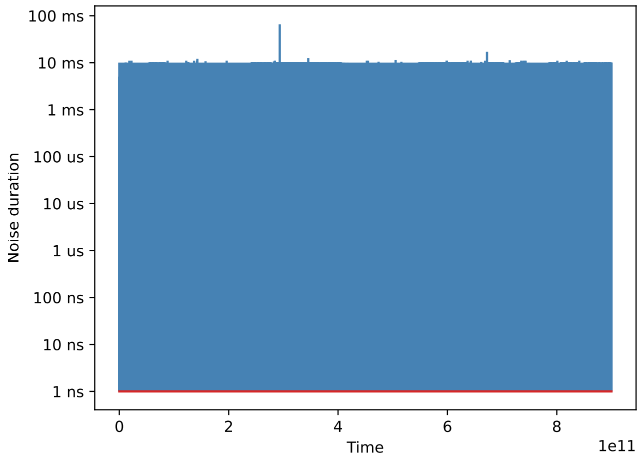
WITH RESOURCE PARTITIONING

- On Stria, the `podman` installation uses `cgroups v1` and `runc`. However, root access is required to use `cgroups v1` to partition resources with control groups.
- Using `cgroups v2` to run rootless containers (on a standalone Linux machine) but was unable to figure out how to get the resource limits to actually take effect
- So…we ran our partitioning experiments on a standalone Linux workstation as root using `cgroups v1`.
- The only resource we partitioned was the CPU
- We considered three different mechanisms for allocating CPU resources in podman
  - `--cpu-quota` & `--cpu-period`
  - `--cpu-shares`
  - `--cpuset-cpus`

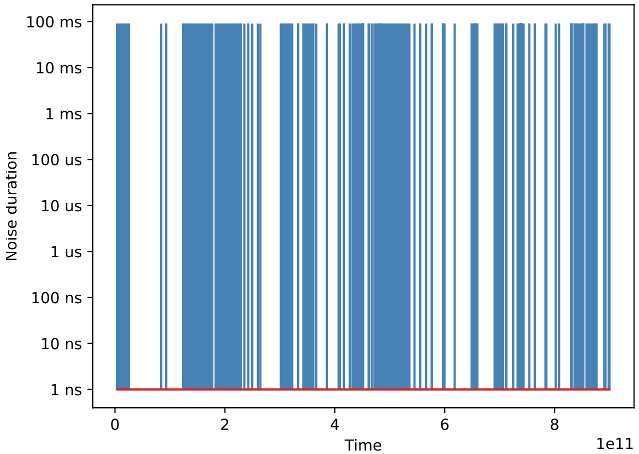
CPU partitioning with `--cpu-quota` and `--cpu-period`

- Using this approach, we can assign a container a **quota** within each **period**. When the container's quota has been exhausted it will not be scheduled again until the current period expires.
- We consider four different periods
  - 100ms
  - 50ms
  - 10ms
  - 5ms
- …and three different quota pairs
  - 90% + 10% (we couldn't run this combination w/ 5ms period; minimum quota is 1ms)
  - 75% + 25%
  - 50% + 50%

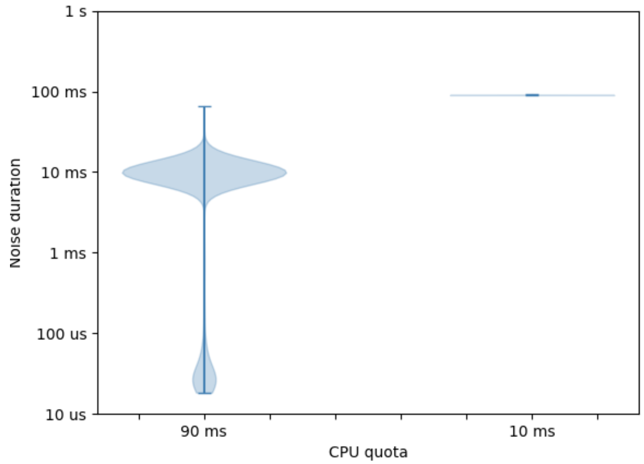
CPU partitioning with `--cpu-quota` and `--cpu-period` (cont'd)



90 ms CPU quota

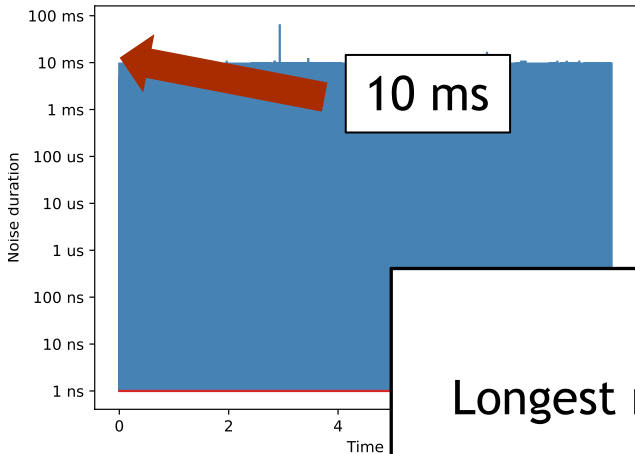


10 ms CPU quota

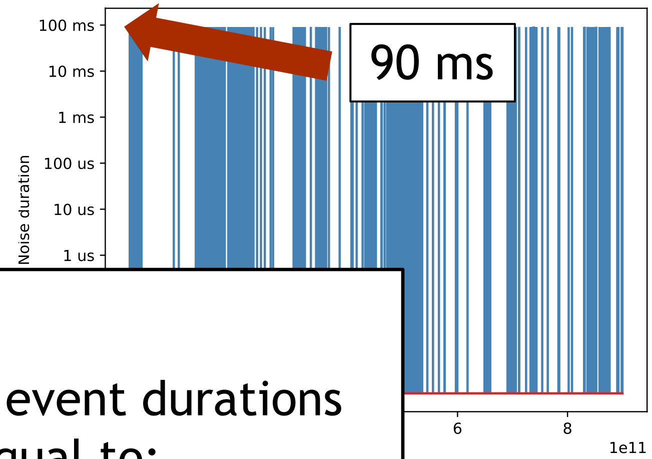


90ms vs 10 ms CPU quota
noise duration distributions

CPU partitioning with `--cpu-quota` and `--cpu-period` (cont'd)

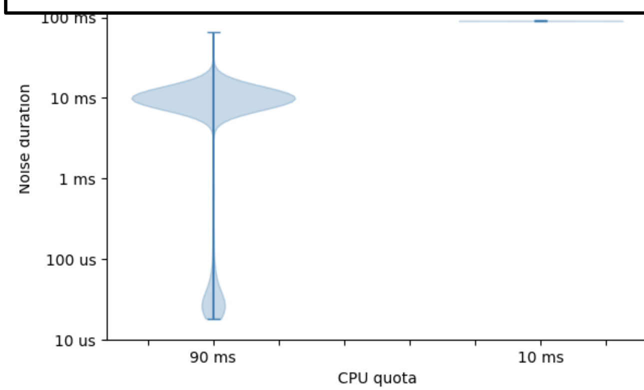


90 ms CP



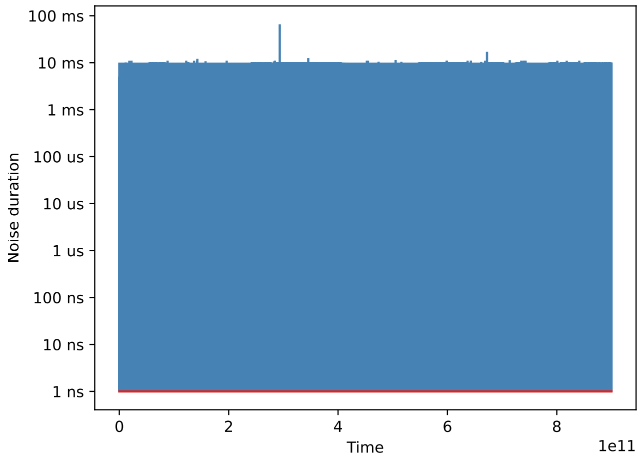
U quota

Longest noise event durations are equal to:
`(period - quota)`

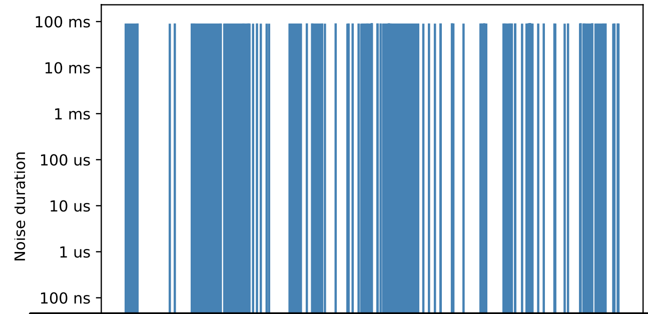


90ms vs 10 ms CPU quota noise duration distributions

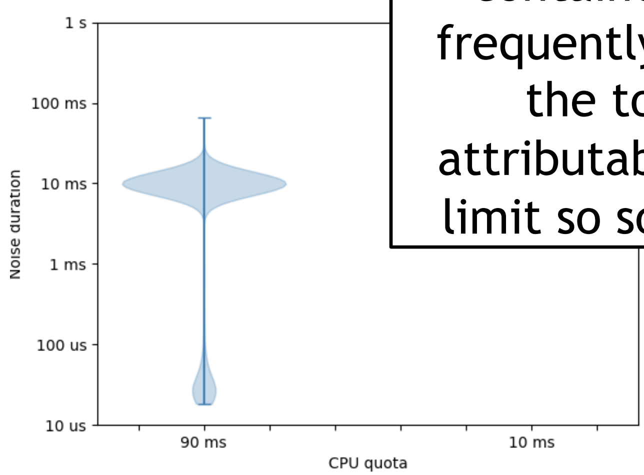
CPU partitioning with `--cpu-quota` and `--cpu-period` (cont'd)



90 ms CPU quota



With 10 ms CPU quota, many fewer noise events are recorded because the container runs much less frequently. More than 1% of the total events are attributable to the resource limit so some are excluded.



90ms vs 10 ms CPU quota noise duration distributions

CPU partitioning with `--cpu-shares`

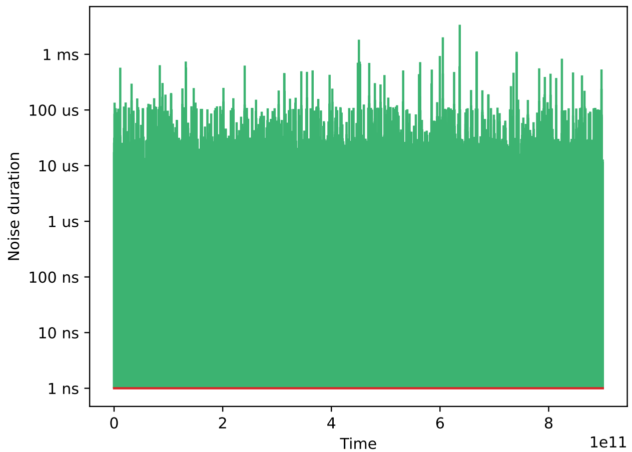
- Using this approach, we can assign a container **shares** of the CPU which are used to prioritize access.

- The Podman documentation states that limiting the number of CPU shares does not prevent each container from using 100% of a CPU provided there are enough CPUs to support all of the processes in all of the containers (i.e., there's not contention for CPU resources)

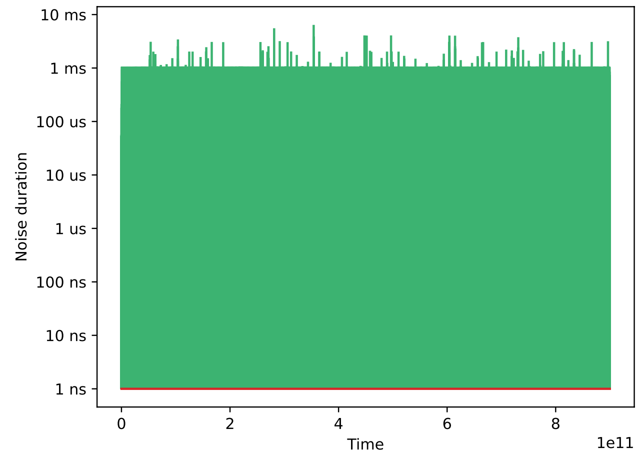
- By default, each container gets 1024 shares

- We consider four configurations:
  - 1024 & 1024
  - 1024 & 512
  - 1024 & 256
  - 1024 & 128

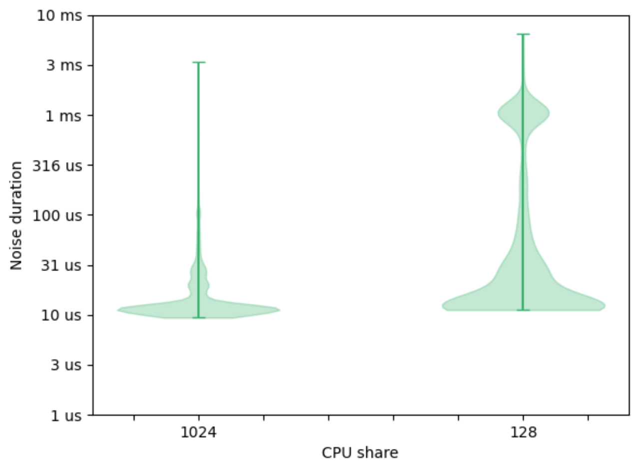
CPU partitioning with `--cpu-shares` (cont'd)



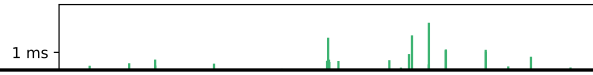
1024



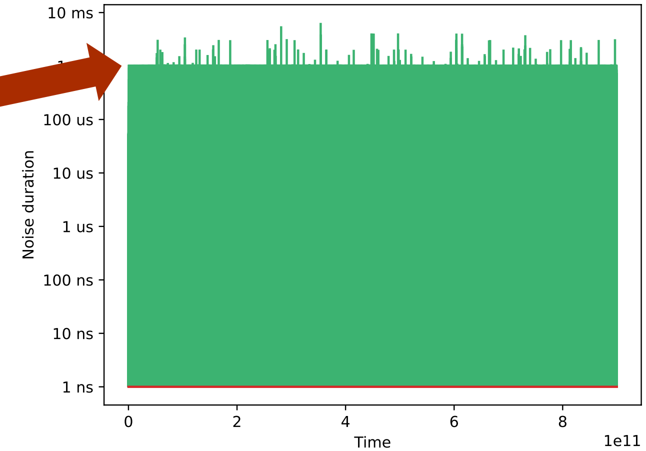
128



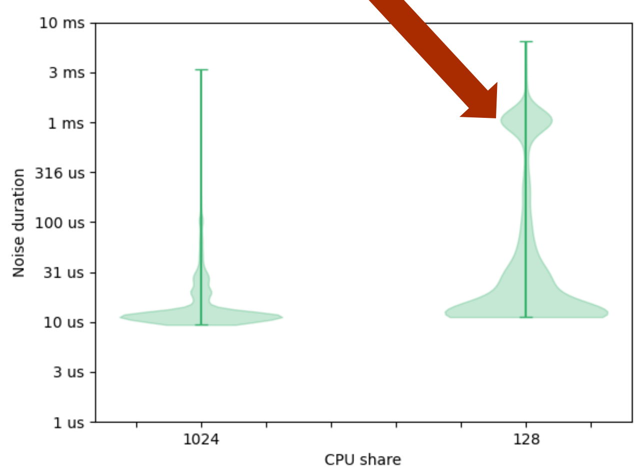
CPU partitioning with `--cpu-shares` (cont'd)



These experiments were performed on a 4-core machine that was largely idle other than these experiments. The Podman documentation suggests that each container should be allocated 100% of one CPU, but there's still a prominent (and mysterious) pattern of ~1 ms noise events for the container with the smaller number of shares



128



CPU partitioning with `--cpuset-cpus`

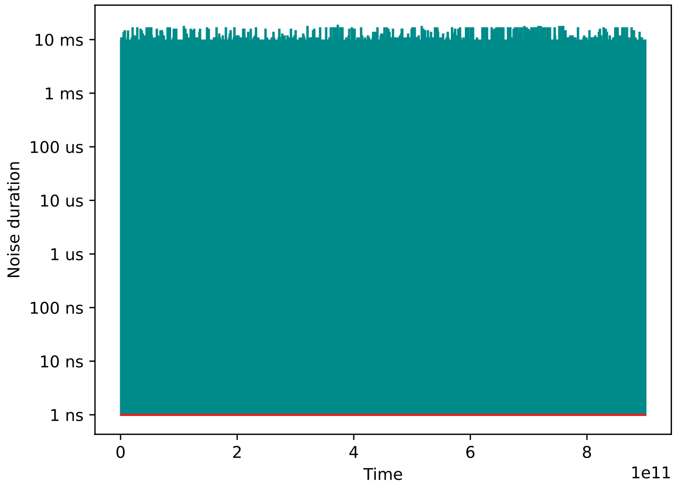
- Using this approach, we can assign a container one or more CPUs

- By default, the operating system is free to run a containers on any available CPU

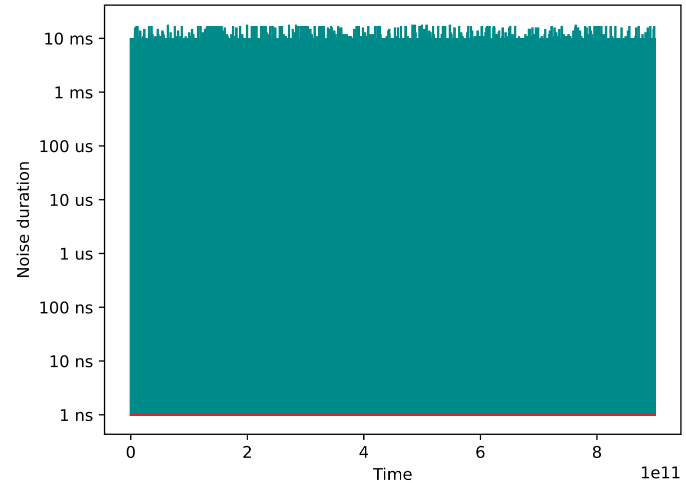
- We consider three configurations:
  - Unpinned (i.e., the containers can run on any available CPU)
  - Pinned to different CPUs
  - Pinned to the same CPU

- We avoid pinning containers to CPU 0 since it tends to have the largest background noise profile

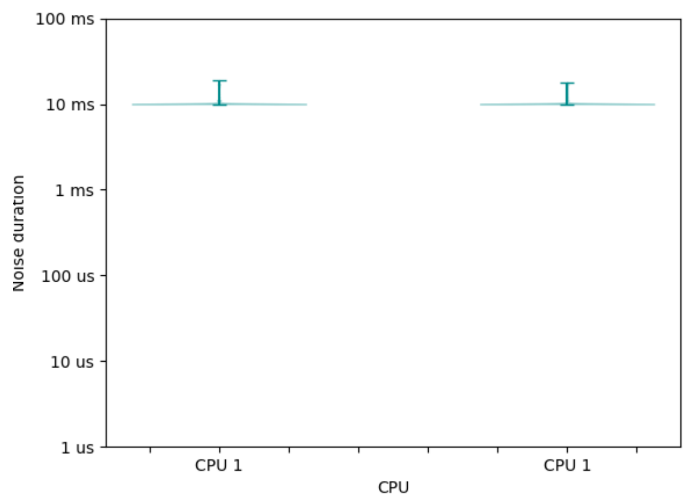
CPU partitioning with `--cpuset-cpus` (cont'd)



First container: CPU1

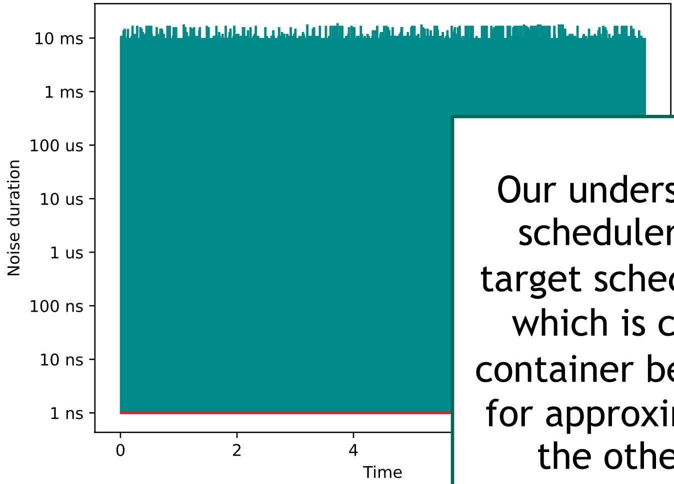


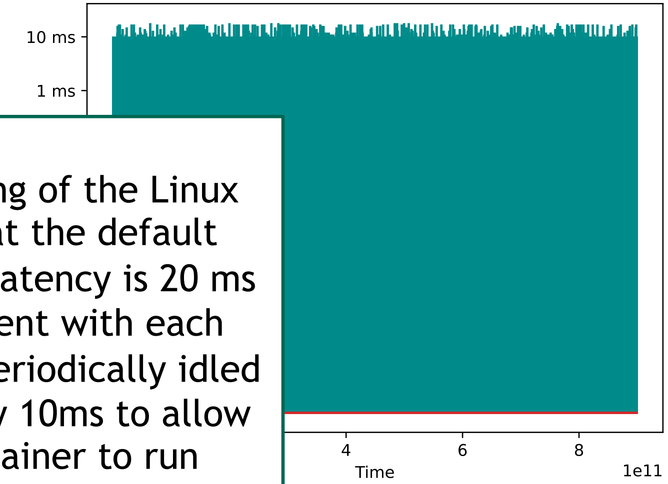
Second container: CPU1



Noise distributions

CPU partitioning with `--cpuset-cpus` (cont'd)

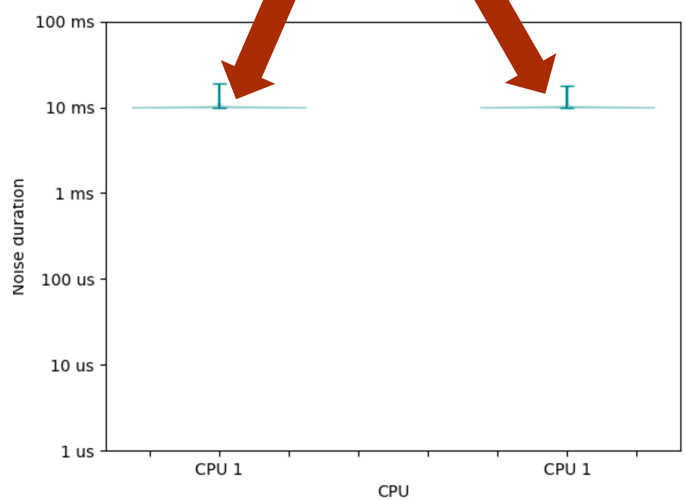




Our understanding of the Linux scheduler is that the default target scheduler latency is 20 ms which is consistent with each container being periodically idled for approximately 10ms to allow the other container to run

First containe

container: CPU1



Noise distributions

Conclusion & Discussion

- We have confirmed that container runtimes are unlikely to impose significant overhead, even on very large systems.
- Partitioning resources between containers using `cgroups` has the potential to introduce significant perturbation into applications.
- Specifically, using `--cpu-quota` and `--cpu-period` to partition CPU resources may introduce perturbation that is likely to degrade the performance of applications as scale increases.
- Given the current state of partitioning tools, better application performance can be obtained by limiting the sharing of hardware resources between containers (e.g., assigning an integral number of nodes/cores to each container).

Acknowledgment

Kurt B. Ferreira
Sandia National Laboratories

Questions?

sllevy@sandia.gov

Sandia National Laboratories is a multi mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.