CS-4365 Final Report

System Resource Utilization Monitoring for Docker Containers

Joseph Azevedo and Bhanu Garg
April 2020

1 Introduction

In recent years, the use of containerization in enterprise systems has skyrocketed as software teams adopt the technology to increase developer agility and simplify deployment. Tools like Docker allow developers to build and deploy their applications in reproducible environments [1], proving to be extremely useful as many companies move their applications to the cloud. Overall industry usage of containerization technologies has jumped from just over half at 55% in 2017 to 87% in 2019 [2], and this trend is unlikely to slow down. With this, many solutions have emerged for monitoring containers, such as Sysdig, which describes itself as a "Linux system exploration and troubleshooting tool with first class support for containers" [3], promising to allow users to closely monitor container system resource utilization. Another is Google's own cAdvisor, which advertises itself as a tool for analyzing "resource usage and performance characteristics of running containers" [4]. While these tools work well for their intended purposes, which range from deep system introspection to cloud-native deployment monitoring, they fall flat when there is a need to analyze resource utilization with both *high granularity* and *low overhead*—something that most tools only provide one of.

An example of a scenario where we need this is when studying the phenomenon of millibottlenecks in containerized systems. First introduced by Pu et al. in The Millibottleneck Theory of Performance Bugs, and Its Experimental Verification, the millibottleneck theory of performance bugs explains that the long tail problem and its underlying cause, millibottlenecks, are caused by very short resource bottlenecks that then propagate through an application system and become amplified due to dependencies and their effect on downstream components [5]. These resource bottlenecks can be caused by a variety of different hardware and software events, such as garbage collection in managed languages or CPU clock speed governors being slow to ramp up during intense bursts of system load. No matter the cause, though, they tend to become aggrandized as the system continues, producing abnormally long response times before the system recovers shortly thereafter and returns to exhibiting standard response times. This results in a so-called "long tail problem," which appears to web service consumers as unusually long response times for a small proportion of requests [6]. Requiring high performance in a scalable system like Google's parallel search system decreases tolerance levels, so these degradations are unacceptable to the companies running these services. In fact, Google reported a loss in up to 20% of revenue from request latencies of 500 milliseconds [7]. Another example of the effect of latency long tail is seen through Amazon sales—– they found that every 100 milliseconds of delay in page loading correlates to a 1% loss of sales [5].

Because these millibottlenecks occur over relatively short intervals (on the scale of tens or hundreds of milliseconds), most monitoring tools are incapable of capturing their effects while simultaneously maintaining low enough overhead as to not impact the measured performance. The situation gets even more complicated when attempting to analyze millibottlenecks in containerized environments, where traditional monitoring methods don't work as expected inside containers. Instead, in order to effectively monitor container resource utilization, you need a custom tool that is able to run on a system and collect information on all running containers. This comprises the main focus of our project, where we investigate various tools for instrumenting Docker containers and ultimately develop our own custom solution in Rust (*see Implementation*).

1.1 Docker

As a technology, containerization provides many benefits to enterprise systems, allowing developers to package applications along with their process-level configuration and dependencies as a single unit, called an image. Then, by using tools such as the Docker runtime, these images can be instantiated as configured processes, called containers, that exist on a host machine as a virtualized environment/runtime. Compared to standard virtualization, containerization is significantly more lightweight, as the Docker runtime directly utilizes the kernel of the host operating system while maintaining a high level of security and isolation [1]. Overall, Docker makes it very easy to deploy applications as containers on a variety of different infrastructures, from managed clouds to on-premise clusters (Figure 1).

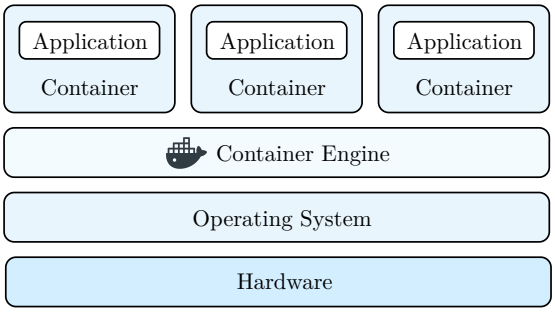

Figure 1: Docker engine architecture, showing multiple applications residing on a single host OS

Nonetheless, containerization presents additional challenges when it comes to monitoring resource utilization for applications. Because it uses virtualization at the operating system level (as opposed to the hardware level), it becomes easy to run multiple containers on one host operating system, better maximizing resource utilization.

This causes many of the resource monitoring techniques used in bare metal or virtualized environments (especially those used to detect and analyze millibottlenecks) to not be effective anymore, as they often operate on the operating system level. For example, Collectl is one of the tools of choice for monitoring CPU, memory, and disk I/O utilization for the Elba project [8] (a research team investigating the emergence and propagation of millibottlenecks), as it is able to record data at a high resolution while maintaining a *low overhead*. Collectl records these statistics for system-wide utilization [9], which, while useful when there is only one application running on a single machine or VM, is much less useful when there are multiple containerized application instances running on one host operating system. The search to find a specialized tool that is able to monitor resource utilization at the level of individual containers became one of the main focuses of our project.

2 Project evolution

The original intent of our project was to adapt the WISE microblog benchmark [10] to run under Docker and Kubernetes, an orchestration platform for managing complex Docker-based deployments [11]. Through this, we hoped to monitor latency delays and potentially discover differences in the ways millibottlenecks emerge in scalable systems.

To accomplish this, we needed a tool that was capable of monitoring the system resource utilization of running Docker containers, and it had to operate with:

1. **high granularity**, which relates to the consistency and precision of collection interval, even under medium and high loads. According to the Nyquist–Shannon Sampling Theorem, we can reliably monitor events of at least 100ms in length with a 50ms monitoring interval [12], which corresponds to the general timescale of millibottlenecks [5].
2. **low overhead**, which relates to the act of collection having a relatively low impact on application performance. This is important to reduce the amount of bias caused by collection in the final results.

These comprised the primary criteria we were using when evaluating potential solutions, and we set out to find a tool that provided both while working on the level of individual containers.

2.1 `docker stats`

The first tool we evaluated for monitoring Kubernetes pods was the built-in Docker resource monitor. While testing the internal Docker CLI tool, `docker stats`, we noticed some pros and cons. `docker stats` is native to Docker, which would require no excessive modification from our end to use in experiments. However, the tool is set to poll the Docker engine every one second for up-to-date usage stats in JSON format, and this interval can not be decreased. Due to the extensively long polling interval, it is impossible to achieve the high granularity required for the project.

2.2 cAdvisor

The next tool we examined was cAdvisor, which is a command line tool developed by Google and geared toward system administrators. It operates as a background daemon, similar to the Docker engine, where it collects statistics about running containers, including "resource isolation parameters, historical resource usage, histograms of complete historical resource usage and network statistics" [4]. However, it was difficult to obtain our desired goals when using it. cAdvisor provides both a web-based GUI tool for viewing statistics and a high-granularity output log. The web-based interface provides useful information, but collects at relatively high intervals around 1 second and is difficult to extract information from. Its output log, on the other hand, provided high granularity, but also monitored much more than just running containers, meaning its overhead would be significantly more than a tool that just instruments containers (Figure 2).

```
1 cName=/system.slice/system-modprobe.slice host=localhost:8086 memory_usage=0
  ↪ memory_working_set=0 rx_bytes=0 rx_errors=0 tx_bytes=0 tx_errors=0
  ↪ timestamp=1583873824045024464 cpu_cumulative_usage=1435685
2 cName=/system.slice/system-systemd\x2dbacklight.slice host=localhost:8086
  ↪ rx_errors=0 tx_bytes=0 tx_errors=0 timestamp=1583873824045832372
  ↪ cpu_cumulative_usage=8054626 memory_usage=20480 memory_working_set=20480
  ↪ rx_bytes=0
3 cName=/system.slice/systemd-logind.service host=localhost:8086
  ↪ timestamp=1583873824056061533 cpu_cumulative_usage=4974510763
  ↪ memory_usage=1798144 memory_working_set=1564672 rx_bytes=0 rx_errors=0
  ↪ tx_bytes=0 tx_errors=0
```

Figure 2: Code snippet showing data collected using cAdvisor. The output is excessive, showing statistics collected for cgroups completely unrelated to Docker.

2.3 Sysdig

Sysdig is another tool built toward letting system administrators monitor fine-tuned performance of a single machine, providing high granularity but coming at the cost of high overhead. Sysdig, unlike the internal Docker CLI tool, allowed for adjustable collection intervals, but had a lower limit at 100ms [13]. Since our target collection frequency was 50ms, we had to make a short patch and recompile the binary to remove the restriction (at elba-docker/sysdig). The base CLI tool, `sysdig`, outputs all syscalls, which is significantly more granularity than we need and is not useful for our monitoring purposes. In addition, intercepting all syscalls introduces significant overhead that we needed to avoid in our monitoring tool (Figure 3).

Sysdig also offers a curses-based CLI tool (Figure 4), `csysdig`, but it is unclear how to translate the given outputs to a log file, which is needed to later analyze how millibottlenecks emerge in and affect an online system. Moreover, despite changing the collection interval to 50 milliseconds with the command-line flag `-d 50`, the resultant display only updated the CPU utilization each second, suggesting that there is a hidden limit to how much granularity the tool provides.

```
39207 21:32:09.180237594 16 ls (9425) > read
↪  fd=3(<f>/lib/x86_64-linux-gnu/libpcre.so.3) size=832
39208 21:32:09.180239174 16 ls (9425) < read res=832 data=.ELF..............>.....
↪  .......@.......8...........@.8...@.....................
39209 21:32:09.180240590 16 ls (9425) > fstat
↪  fd=3(<f>/lib/x86_64-linux-gnu/libpcre.so.3)
39210 21:32:09.180242123 16 ls (9425) < fstat res=0
39211 21:32:09.180243743 16 ls (9425) > mmap addr=0 length=2560264
↪  prot=5(PROT_READ|PROT_EXEC) flags=1026(MAP_PRIVATE|MAP_DENYWRITE)
↪  fd=3(<f>/lib/x86_64-linux-gnu/libpcre.so.3) offset=0
39212 21:32:09.180247105 16 ls (9425) < mmap res=7F7139203000 vm_size=9224
↪  vm_rss=348 vm_swap=0
39213 21:32:09.180248119 16 ls (9425) > mprotect
39214 21:32:09.180251421 16 ls (9425) < mprotect
```

Figure 3: Code snippet showing data collected using `sysdig container.image=ubuntu`. The output shows all syscalls related to a single container, which, while informative, is undesired for our use case.

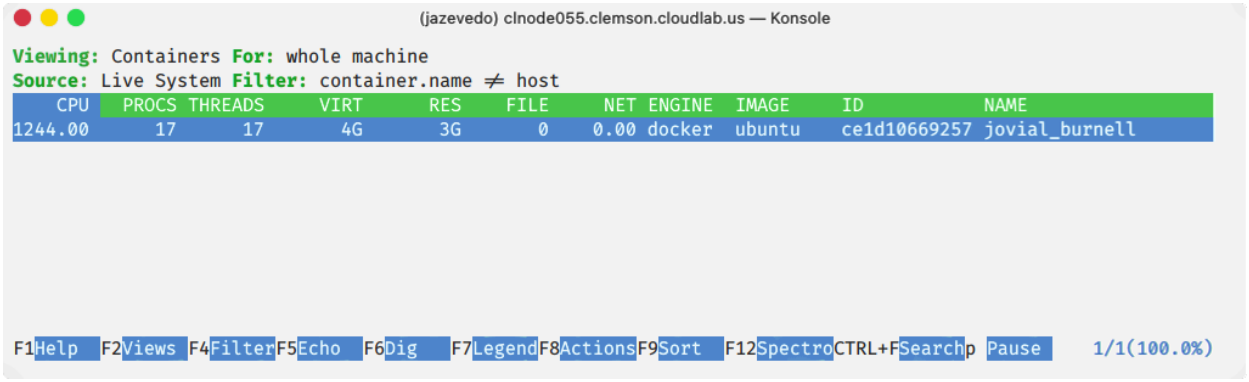

Figure 4: Example output of the `csysdig` monitoring tool in container mode, showing its curses-based interface. The command used is `csysdig -vcontainers`.

Ultimately, we were not able to find an existing tool that provided the necessary combination of *high granularity* and *low overhead*, so we looked into modifying an existing tool.

2.4 Moby

Moby is an open-source framework that is the core of the Docker engine and is built in Golang. It runs as a daemon on the host system where the Docker CLI tool communicates with it through Unix sockets. Because it already contained a backend implementation for `docker stats`, which almost satisfied our requirements but lacked granularity, we looked toward modifying Moby to increase its collection granularity. After forking Moby on GitHub, we added a configuration option for the `docker stats` collection interval to control the collection granularity. We then proceeded to add functionality that outputs utilization logs for all running containers and changed it to output CSV instead of JSON. Although we were able to make adjustments to Moby, there were still drawbacks that prevented us from using it. Moby presented difficulties in measuring overhead due to it being a part of the core Docker runtime, which meant there was no control for users to start and stop it because it is always running. Our patch also adds complexity to already-complex software and can not be easily generalized to include Kubernetes metadata, one of the goals of our project at that time. Another concern with Moby was the difficulty involved with maintaining a fork that could face problems with security patches in the future. Finally, Moby faces inherent problems due to being built in Golang, which, while much faster than other interpreted languages like Python, still uses a built-in garbage collector, leading to unpredictable and inconsistent resource utilization in

the instrument itself.

Due to the limitations faced with Moby, we decided to build our own tool called rAdvisor (a play on Google's cAdvisor), a Rust-based tool designed to achieve *high granularity* and *low overhead*. From this, we were able to compare its performance with Moby to evaluate whether it achieves its goal and has the potential to be a useful tool in detecting and analyzing millibottlenecks in containerized environments.

3 Implementation

3.1 Moby

Moby, as discussed earlier, is the Docker core engine (built in Golang) which we forked to elba-docker/moby and modified to add custom collection configuration and behavior. The first modification was to use the existing configuration framework that loads the daemon configuration from a JSON file at `/etc/docker/daemon.json` (Figure 5). Afterward, we replaced the hard-coded collection interval for `docker stats` (which was originally 1 second) with the value loaded from the config (in milliseconds). By doing this, we were able to adjust the collection granularity at runtime by changing a single line in the config file, instead of having to recompile the engine from source.

```
1  {
2    "exec-opts": ["native.cgroupdriver=systemd"],
3    "log-driver": "json-file",
4    "log-opts": {
5      "max-size": "100m"
6    },
7    "storage-driver": "overlay2",
8    "stats-interval": 50
9  }
```

Figure 5: Sample Docker daemon configuration used in our experiments (source), showing the added configuration option to change the `docker stats` collection interval (line 8)

Even still, there were more improvements to be made. For one, `docker stats` uses a long-lived HTTP connection to stream stats that is unnecessary in our case and needs to be started/stopped either manually or by some other tool each time the set of running containers changes. To alleviate this, we introduced an automatic collection extension that leverages one of Golang's first-class concurrency control units, Goroutines [14]. These functions, which act as light-weight threads, allow us to dispatch a new collector every time a new container starts that runs in the background, getting stopped when the associated container finally stops. Then, we re-use the streaming functionality implemented in `docker stats` and redirect the output to write to a buffered writer in front of a file at `/var/logs/docker/stats/{id}.log`. The buffered writer is necessary to reduce the frequency of writes at runtime, which lowers the overhead of collection (a similar technique is employed in the Elba project's WISETrace, which intercepts and logs network syscalls as a kernel module [15]).

Another performance-increasing modification we made is switching from the JSON stats serializer used for the `docker stats` API backend to a more basic serializer that uses CSV. This resulted in a speed increase when marshalling the statistics struct by a factor of 2.335 (Table 1).

These functional changes can be summarized by Figure 6, a sequence diagram showing the lifecycle of container statistics collection.

Table 1: Benchmark (source) results for serializing a single `StatsJSON` struct using both JSON (original) and CSV (modified) targets. The test was ran on Ubuntu 20.04 LTS with an Intel Pentium G3258 and 8GB RAM.

Test	**Number of operations**	**Execution speed (ops/sec)**
JSON Serialization	883,716	27,475
CSV Serialization	2,034,567	11,769

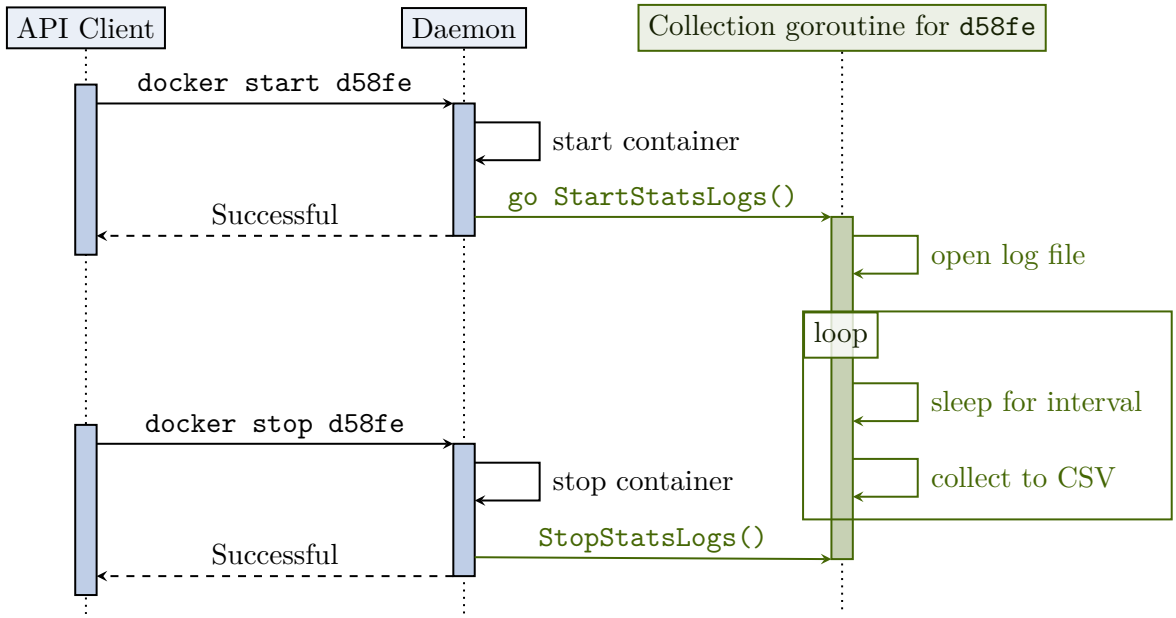

Figure 6: Simplified sequence diagram of the modifications made to Moby to allow for fine-grained container monitoring. Additions we made to the Moby engine are highlighted in green.

3.1.1 Packaging

To deploy our patched version of Moby, we used the existing build tools in docker/docker-ce (the repository used to build the standard Docker packages) to build our own `.deb` packages for Docker and Docker CLI. We then used the generated packages (*see Appendix A*) to deploy our patched version in our experiments.

3.2 rAdvisor

With the limitations we observed in our patched version of Moby, we created rAdvisor to attempt to overcome them. Our goal with rAdivsor was to create a tool that collects resource utilization statistics on all running containers and outputs it in a useful format, similar to Collectl. During development, we sought to specifically address the shortcomings of Moby in the way we designed rAdvisor. For one, it is an extremely focused tool—its sole purpose is to instrument running containers—so it is easier to debug, maintain, and limit the performance impact of. This also means it is much easier to measure its overhead, which we attempted to do via experimental observation (*see Evaluation*). Its source code can be found in its corresponding repository on GitHub: elba-docker/radvisor.

The tool was built using the Rust language, which is a statically-typed, C-like language intended for memory-safe programming. Our reasoning behind picking Rust was because it is a low level language with similar performance to C, ultimately allowing for lower overhead [16]. Compared

to Golang, Rust has no garbage collector, allowing for greater consistency in sampling due to the lack of short resource usage peaks that themselves have been shown to cause short-lived resource bottlenecks [5]. It also provides many benefits over C, such as its purpose-built type system and compiler that are able to effectively eliminate runtime memory errors.

3.2.1 Structure

rAdvisor runs in the background and collects statistics for active containers at a configurable collection interval. The tool leverages the advanced concurrency capabilities of Rust, utilizing two main threads (Figure 7) that accomplish its purpose by communicating across a single, shared channel:

1. The **polling thread** polls the Docker daemon API through a Unix socket at a configurable interval to get a list of active and running containers. The thread then filters and processes the list, and sends events (add or remove) to the collection thread utilizing the cross-thread channel.

2. The **collection thread** runs at a configurable interval and reads a set of files in the cgroupfs pseudo-filesystem using Linux cgroups. The thread then writes their contents to a buffered CSV writer that eventually writes to a file in `/var/log/radvisor/stats` each time the buffer is flushed. This is done for every container in the list of active containers, which is determined by the incoming events sent from the polling thread. The collection interval is separate from the polling interval, and is generally much shorter (the default is 50 milliseconds versus 1 second for the polling thread). This is because the list of active containers changes fairly infrequently, whereas the goal of the tool is to collect container resource utilization with *high granularity*.

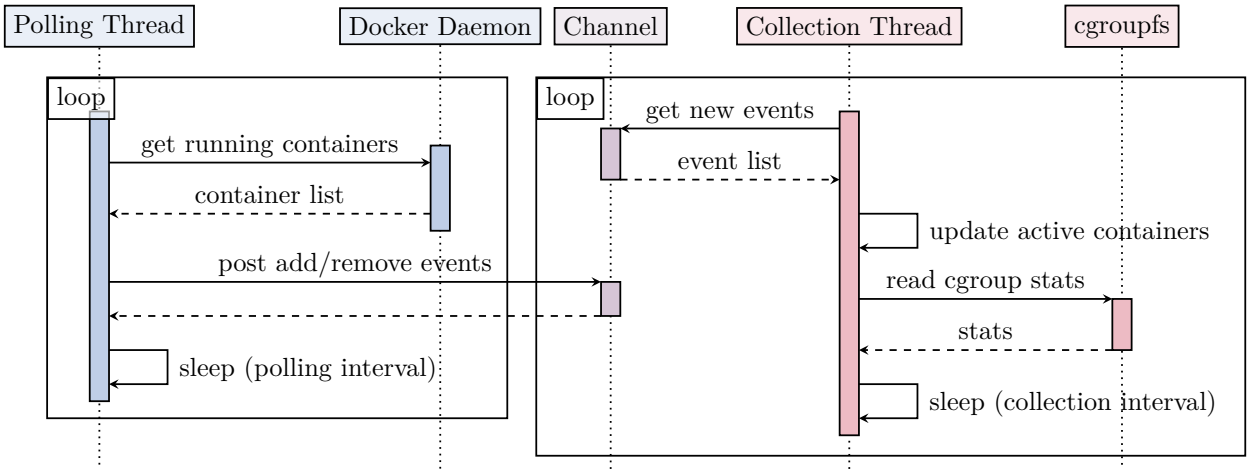

Figure 7: Simplified sequence diagram of the concurrent architecture of rAdvisor. Operations related to the *polling thread* are blue; operations related to the *collection thread* are red.

3.2.2 Implementation Details

As mentioned before, the implementation of the collection functionality heavily relies on Linux cgroups. Added to the Linux kernel version 2.6.24 in 2008 [17], cgroups are one of the core technologies that make operating system-level process isolation such as Docker containers possible [18]. They provide capabilities to limit and monitor resource utilization (specifically, CPU, memory, and block I/O) [19] in hierarchical groups of processes. In its standard operation, Docker creates a new cgroup for each running container and adds all child processes to it [19], letting it control resource allocation for the container as a whole.

While the normal operation of the Docker runtime primarily leverages the isolation and resource

limiting capabilities, rAdvisor exclusively uses the monitoring capabilities cgroups also provide. Within each cgroup's location in the cgroupfs pseudo-filesystem is a set of files that contain information on the current resource utilization of all processes in that cgroup. For example, the `cpuacct.usage_percpu` file contains information on the CPU usage of the cgroup broken down by logical cores (Figure 8). At its most basic level, rAdvisor's main task is to read a subset of these files for each container as bytes and copy the bytes to a buffer that eventually writes to a CSV file. More information can be found about the specific cgroup subsystems and files used by rAdvisor in the corresponding documentation.

```
10988262282 10955397365 11420884004 12532674907 11310602969 12382279847 12193108713
↪ 10432778271
```

Figure 8: Example `cpuacct.usage_percpu` file taken from an eight-core Linux system. Each number corresponds to the CPU time (in nanoseconds) spent by processes in the cgroup for the corresponding core.

In addition to the information provided by cgroups, rAdvisor also leverages the metadata it receives from the Docker API. It does this by including its own collection metadata at the top of each file in YAML format [20], which, in addition to the container metadata, includes information on the target cgroup, poll time, and system information (useful for tagging collected statistics with metadata about the system when later analyzing experimental results). This combination of YAML metadata and CSV content is called CSVY [21], and is the primary output format of rAdvisor because of its ability to combine tabular data with structured metadata (Figure 9).

Another important implementation detail of rAdvisor is it's leveraging of Rust's memory model. As with any statically compiled language with no garbage collector, Rust requires programs to manage the memory they use, which can exist either on the stack or the heap [16]. Rust has the concept of "borrows" to memory which provide a memory-safe abstraction over pointers from other languages, letting us pass around references to stack variables in lieu of locally allocating heap memory where it is used. An area where this is especially leveraged is with a set of working buffers used for copying (and performing basic parsing on) stats collected from cgroups files to the internal CSV file buffer. The same set of working buffers (and underlying memory) exists on the stack of the main method of the thread and exists throughout the entire lifetime of the program, saving countless heap allocations that would likely be used if rAdvisor was written in a managed language. The same pattern is applied to the entirety of the *collection thread*, resulting in almost no heap allocations at runtime on that thread. This is especially important because it runs at the highest frequency and needs to operate with as low overhead as possible, which allocations would work against.

This ultimately helps to lower the runtime footprint of rAdvisor in its critical collection path, contributing to its ultimate goals of *high granularity* and *low overhead*.

3.2.3 Target providers

rAdvisor was designed with modularity in mind, largely because we were not sure if we would be continuing pursuing our original Kubernetes-oriented project when we were developing it. Because of this, it actually includes two separate implementations for obtaining *collection targets*, which are essentially cgroups that we are interested in obtaining stats for: Docker containers (as discussed earlier) and Kubernetes pods. Both *target providers* operate in similar ways, and their differences stop once the add/remove events are pushed to the collection thread. In this way, the two aspects of the program, target selection and statistics collection, remain logically separate and independent. This contributes to the maintainability of the tool and provides the opportunity to re-use its functionality for other purposes (such as monitoring specific processes).

```
---
Version: 1.1.7
Provider: docker
Metadata:
  Created: "2020-04-23T05:07:25Z"
  Command: "bash -c 'stress --cpu 2 --vm 2 --vm-bytes 128M --timeout 240s'"
  Id: 188b184eea7862d088e361c6ef7c094466ef119623938b72fb2a59
  Image: "jazevedo6/direct_collectl:v1.0"
  Labels: {}
  Names:
    - /practical_engelbart
  Ports: []
  Status: Up 4 seconds
  SizeRw: ~
  SizeRootFs: ~
System:
  OsType: Linux
  OsRelease: 4.15.0-88-generic
  Distribution:
    Id: ubuntu
    IdLike: debian
    Name: Ubuntu
    PrettyName: Ubuntu 18.04.1 LTS
    Version: 18.04.1 LTS (Bionic Beaver)
    VersionId: "18.04"
    VersionCodename: bionic
    CpeName: ~
    BuildId: ~
    Variant: ~
  MemoryTotal: 264061528
  SwapTotal: 3145724
  Hostname: node-2.d-rc-100-01.infosphere-pg0.clemson.cloudlab.us
  CpuCount: 40
  CpuOnlineCount: 4
  CpuSpeed: 2200
Cgroup:
↪ system.slice/docker-188b184eea7862d088e361c6ef7c094466ef119623938b72fb2a59.scope
CgroupDriver: systemd
InitializedAt: 1587618451218168894
---
read,pids.current,pids.max,cpu.usage.total,cpu.usage.system,cpu.usage.user,
↪ cpu.usage.percpu,cpu.stat.user,cpu.stat.system,cpu.throttling.periods,
↪ cpu.throttling.throttled.count,cpu.throttling.throttled.time,
↪ memory.usage.current,memory.usage.max,memory.limit.hard,memory.limit.soft,
↪ memory.failcnt,memory.hierarchical_limit.memory,
↪ memory.hierarchical_limit.memoryswap,memory.cache,memory.rss.all,memory.rss.huge,
↪ memory.mapped,memory.swap,memory.paged.in,memory.paged.out,memory.fault.total,
↪ memory.fault.major,memory.anon.inactive,memory.anon.active,memory.file.inactive,
↪ memory.file.active,memory.unevictable,blkio.service.bytes,blkio.service.ios,
↪ blkio.service.time,blkio.queued,blkio.wait,blkio.merged,blkio.time,blkio.sectors
1587618451239134986,2,6143,255777313,0,255777313,10278936 5194252 230498788 9805337
↪ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
↪ 0,23,2,8,0,0,1667072,7086080,4294967296,9223372036854771712,0,4294967296,,0,
↪ 352256,0,0,,1758,1672,2154,0,0,352256,0,0,0,Total 0,Total 0,Total 0,Total
↪ 0,Total 0,Total 0,,
```


Figure 9: Example (abridged) output file taken from one of our experiments on CloudLab (*see Evaluation*), showing the CSV data combined with the YAML metadata (with container metadata and system information)

4 Evaluation

During our evaluation phase, our main goal was to compare our two potential tools for instrumenting Docker containers: our patched version of Moby and rAdvisor. With this, we wanted to compare how they succeed (or fail) in collecting container statistics with **high granularity**—being able to collect with a low interval and to do so consistently—and **low overhead**—having a low, consistent impact on measured performance.

To accomplish this, we performed experiments on CloudLab, an experiment testbed for cloud computing research [22], [23]. Using CloudLab, we can provision a set of physical hosts with which to run benchmarks on while running rAdvisor or Moby in the background. These experiments are then controlled by a small handful of experimental workflow scripts, which are lengthy Bash scripts full of `ssh` (remote terminal) and `scp` (remote file transfer) commands used to orchestrate running benchmarks on remote cloud nodes. At the end, these scripts transfer results off of the remote hosts, which are composed of logfiles, metadata, and configuration values bundled in a single archive.

With these scripts (and an automation tool), we are able to run hundreds of replicas of these experiments, which can use anywhere from 1 to 10 remote nodes at a time. Finally, we can analyze the experimental data and produce graphs and tables as part of our analysis (*see Results*).

4.1 Test types

For our experiments, we used 3 different test types designed to be a combination of synthetic and real-world tests. In each, we are able to measure the direct and indirect performance overhead of rAdvisor and Moby, as well as investigate whether they are able to maintain *high granularity*, even under medium/high load:

1. **Direct overhead** (`d-` test ID prefix) - In this test, we run a synthetic system stressor to consume system resources in order to see how well rAdvisor and Moby respond to load. Specifically, we use `stress` to impose memory, disk, and CPU stress on the system, which describes itself as "a deliberately simple workload generator for POSIX systems" [24]. We use it because it is simple to run and can be run in multiple Docker containers at once. In essence, we're trying to examine whether the tools' collection degrades at medium/high CPU utilization, which is part of our *high granularity* criterion (it should not degrade). Additionally, we're looking what the direct performance impact of running rAdvisor and Moby is on the overall system CPU usage, and whether it's significant. This is part of our *low overhead* criterion, and we test it at two different levels of system load: 50% peak stressor CPU load and 100% (controlled by setting cgroup-based limits on CPU resources for the stressor containers). These allow us to test both medium and high synthetic workloads, respectively (Figure 10).

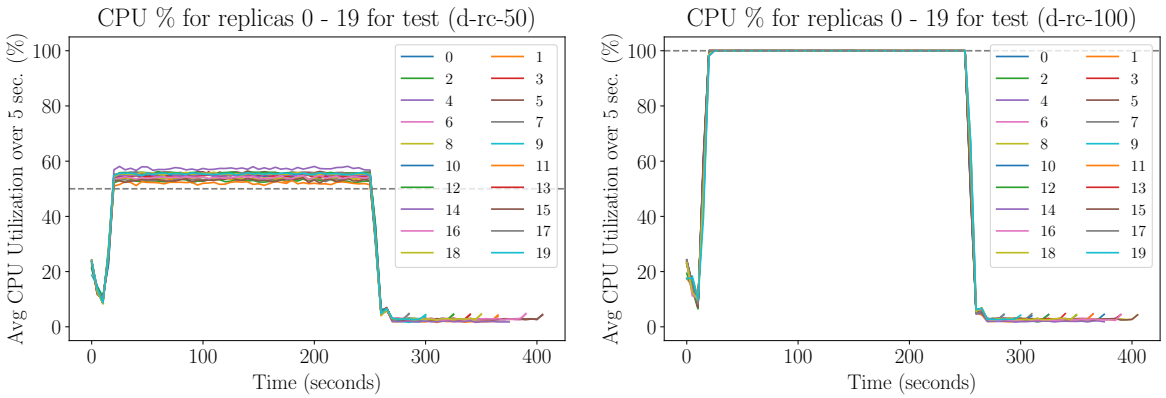

Figure 10: Average CPU utilization graphed for the two different peak stressor CPU load configurations, showing the 50% peak CPU utilization (left) test and the 100% test (right)

2. **Indirect overhead (synthetic)** (`i-` test ID prefix) - In the second test, we sought to perform similar evaluation as the direct overhead experiment, as well as to see what the performance impact of running rAdvisor and Moby is in CPU throughput. To do this, we use nbench, a synthetic benchmark developed by the now-defunct BYTE magazine in the 1990s designed to measure integer, floating point, and memory performance [25]. Because the original software wasn't designed to run on Linux, we use a Linux port developed by Dr. Uwe Mayer in 1996 [26] that can be found on GitHub at elba-docker/nbench. The benchmark outputs both per-test scores and overall integer, floating point, and memory aggregate scores that we can use to directly compare experimental configurations running rAdvisor, patched Moby, or neither (*See Experimental matrices*). In these tests, we also test the configurations at the same two levels of system load as in the first test, 50% and 100%

3. **Indirect overhead (response time)** (`ii-` test ID prefix) - In our third and final test configuration, we run a real-world application benchmark using the WISE microblog benchmark developed by the Elba project (with minor modifications, *see Appendix A*). In it, a sample microservices-based application is used that resembles a microblog like Twitter [10] (Figure 12), which was adapted to operate in a containerized environment. As much as possible, the workloads are designed to simulate real-world enterprise workloads, using a interaction duration model based on Gaussian distributions as well as a stochastic model for determining the next interaction to be performed [10]. The goal of using it is to see the indirect performance impact of running rAdvisor and Moby on an online application's performance, measured by the response time distribution observed for each configuration. By doing this, we are able to examine the overhead in a real-world application test that is also in the target domain of the Elba project. For this test, instead of varying the peak CPU utilization, we apply both a sustained and a bursty workload (Figure 11) and observe the performance under each.

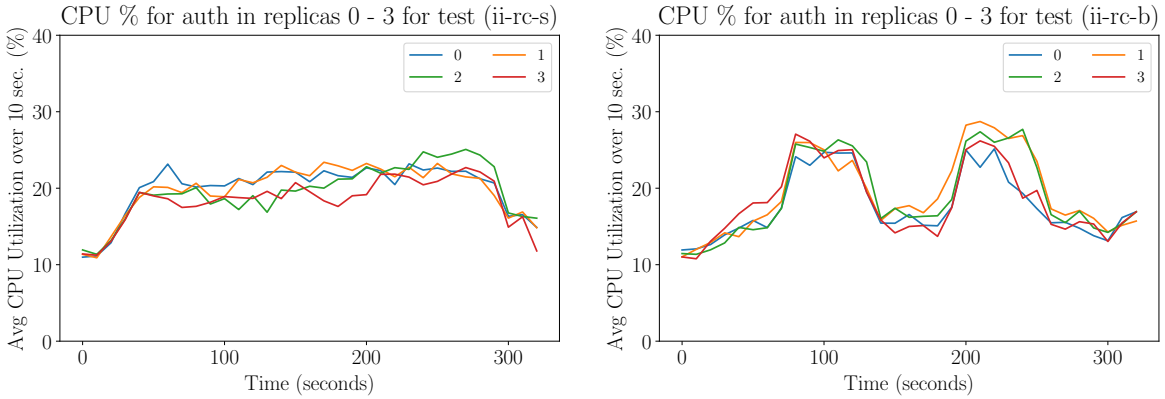

Figure 11: Average CPU utilization graphed for 4 test replicas in each of the workload configurations: sustained (left) and bursty (right). The workloads leverage the statistical workload model to produce semi-random sets of requests during the course of the experiments.

4.1.1 Experimental matrices

In addition to varying specific parameters of the test configurations (like whether they impose 50% or 100% peak CPU load or use a bursty/sustained workload) (Figure 13), we also tested each configuration against another common dimension of independent variable: enabled instrumentation. Across each test type (`d-`, `i-`, and `ii-`), we tested the following combinations of enabled instrumentation:

- **r** - rAdvisor
- **rc** - rAdvisor and Collectl
- **c** - Collectl
- **mc** - patched Moby and Collectl

- **m** - patched Moby

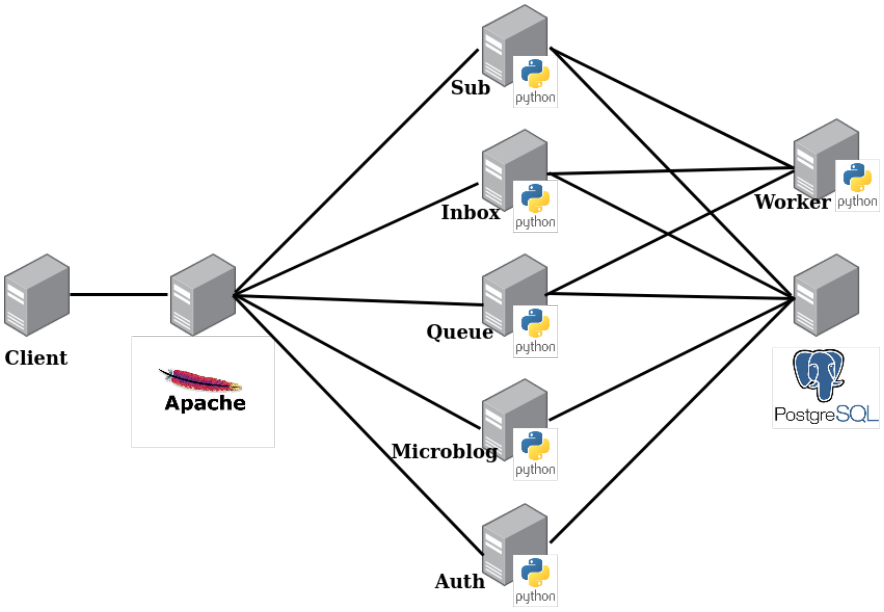

Figure 12: Overall microservices-based architecture of the microblog benchmark application, used in the indirect overhead (response time) experiment configuration

```
# Whether to use the patched docker version; either 0 or 1
readonly USE_PATCHED_DOCKER=1
# Whether to enable rAdvisor; either 0 or 1
readonly ENABLE_RADVISOR=0
# Whether to enable collectl; either 0 or 1
readonly ENABLE_COLLECTL=0
```

Figure 13: An excerpt from the template `config.sh` file for one of the experiments, showing the boolean flags that are used to control the enables instrumentation in the final experiment runs

These combine to form a 5×2 matrix of independent variable combinations, which we are able to then run on CloudLab using our automation tool.

4.2 Tooling

To conduct our experiments, we use a variety of custom-built tooling to assist in experiment conduction, automation, and result analysis.

4.2.1 Experimental workflow scripts

The experimental workflow scripts are a collection of comprehensive Bash scripts (at elba-docker/experiment) that install dependencies and run experiments on CloudLab. They do this by running on a remote host (called the orchestrator) that then uses `ssh` and `scp` to control an array of experiment nodes, additionally managing data collection and tear-down at the end. The scripts are utilized by the automation scripts for completely automated experiment configuration, execution, and data collection.

4.2.2 Automation scripts

One challenge we encountered while running our experiments is that setting up and provisioning experiments on CloudLab is tedious and takes an excessive amount of time. Because of this, we developed an automation script (at elba-docker/automation) that would handle experiment

execution, from configuration creation to results extraction.

At a high level, the automation script performs the following tasks:

1. Reads a config file (Figure 14) and generates a set of test runs from it, expanding experiment matrices to include all possible combinations.

```
tests:
    # Indirect overhead using application benchmark (microblog)
  - id: ii
    replicas: 25
    experiment: indirect_response_time
    profile: MicroblogBareMetalC8220
    options:
      HOSTS_TYPE: physical
      HARDWARE_TYPE: c8220
    matrix:
      - name: workload
        values:
          - id: b
            options:
              WORKLOAD_CONFIG: "conf/bursty.yml"
          - id: s
            options:
              WORKLOAD_CONFIG: "conf/sustained.yml"
      - name: tools
        values:
          - id: rc
            options:
              USE_PATCHED_DOCKER: 0
              ENABLE_RADVISOR: 1
              ENABLE_COLLECTL: 1
            # additional values omitted
```

Figure 14: One of our test configurations from the automation script config file, showing the matrix definition, with two dimensions (`workload` & `tools`) and the number of replicas (25). This results in 250 ($5 \times 2 \times 25$) total test runs that will be performed for this test.

2. Logs into CloudLab using the Selenium browser automation toolkit, which lets us control a browser session from Python and automate clicking/typing in various inputs [27].
3. Provisions an experiment from CloudLab and renders the configuration `config.sh` file to include the test run's configuration parameters.
4. Transfers all necessary files onto the remote orchestrator host, such as SSH keys and the rendered configuration file.
5. Invokes the experimental workflow script on the remote orchestrator host to conduct the experiment, using `pexpect` to automate an `ssh` session [28].
6. Transfers the results archive off of the orchestrator and terminates the CloudLab experiment.

Throughout the lifetime of an experiment run, we utilize a common pattern in Python called retry loops [29], where we attempt to perform an action repeatedly if it fails after a short back-off duration (Figure 15). This allows us to obtain considerable reliability for the automation script despite the unreliable workload it performs, letting us run it over the course of days and have it complete all configured experiments despite any transient errors that may occur during the time interval.

```
[ INFO 13:17:14.815] [ii-rc-s-24] Provisioning new experiment from cloudlab
[ WARN 13:18:20.644] [ii-rc-s-24] Failed to provision experiment with name ii-rc-s-24 on
↪  Cloudlab after the 1st attempt; retrying in 90.0 seconds
[DEBUG 13:18:20.644] [ii-rc-s-24] Caused by:
                                  ElementNotInteractableException('element not interactable\n
                                  ↪  (Session info: headless chrome=81.0.4044.122)', None)
[ INFO 13:20:06.241] [ii-rc-s-24] Instantiating experiment ii-rc-s-24
↪  (9fbbd8df-87c0-11ea-b1eb-e4434b2381fc) profile = MicroblogBareMetalC8220
[DEBUG 13:20:06.266] [ii-rc-s-24] Waiting for experiment to become ready
[ INFO 13:31:51.361] [ii-rc-s-24] Successfully provisioned new experiment from cloudlab:
↪  MicroblogBareMetalC8220 (9fbbd8df-87c0-11ea-b1eb-e4434b2381fc) profile = ii-rc-s-24
                                  | clnode089.clemson.cloudlab.us...
[ INFO 13:31:51.362] [ii-rc-s-24] Using working/ii-rc-s-24 as the working directory
[ INFO 13:31:51.362] [ii-rc-s-24] Wrote rendered config file to working/ii-rc-s-24/config.sh
[ INFO 13:31:51.362] [ii-rc-s-24] Starting execution thread (clnode075.clemson.cloudlab.us)
```

Figure 15: Example output from the automation script, showing it encountering an exception during its browser automation and waiting 90 seconds before trying again. Afterwards, it tries again and succeeds, proceeding on to rendering the configuration and executing the experiment on the remote cloud.

Additionally, by utilizing Python's specialized concurrency capabilities, we are able to run experiments with configurable concurrency (limited by the number of available nodes in CloudLab). By utilizing this script as the core of our experiment execution workflow, we have been able to run over 300 experiments on CloudLab which we use to analyze the effectiveness of our tools.

4.2.3 Parsers and analysis notebooks

The data collected from running experiments is then parsed and aggregated using Python scripts, additionally using Jupyter notebooks at the end for analysis. The scripts consisted of parsers for each kind of log file encountered in our results: nbench, milliScope/WISETrace (`revfrom`, `connect`, and `sendto`), Moby, rAdvisor, and Collectl (`cpu`, `tab` (memory), and `dsk` (disk)). We also utilize an extraction script that runs on a folder of all the archives obtained from running the experiments and recursively extracts them. Additionally, due to the large volume of data involved (over 50 GB when unzipped), we utilized parsers that operated *lazily*: only actually reading the underlying log files during analysis when we first access the data.

Along with the parsers, we also used Jupyter notebooks, which are documents that contain code, visualizations, and Markdown text. They are useful because they allow for shared documents that contain code and equations with a user interface that allows for progressive development and visualization of data . Because Jupyter notebooks allow for easy visualization, they aid us in creating graphs based off of our experimental data (Figure 16).

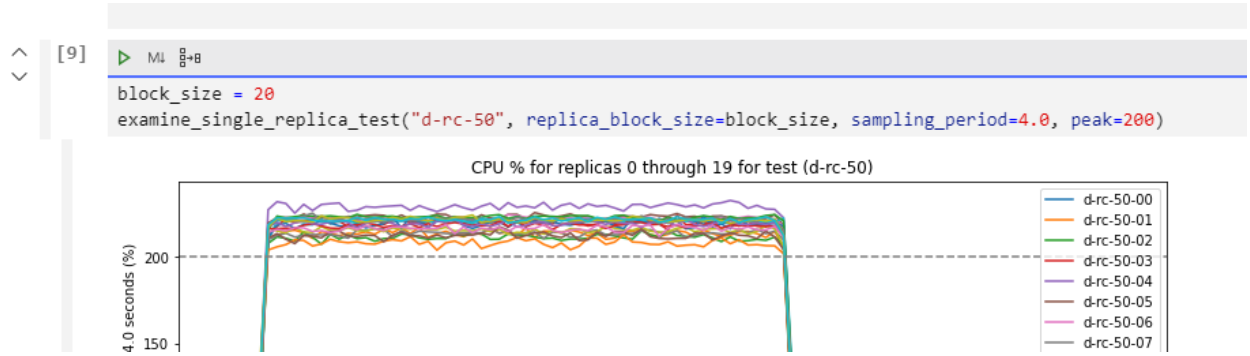

Figure 16: Screenshot of one of our Jupyter notebooks, showing how we can generate graphs and analyze our data in small units of code called *cells*

5 Results

After conducting 30 replicas of each experiment (360 separate CloudLab experiments total, involving 3,660 nodes), we have obtained 5.97 gigabytes (gzip-compressed/archived) of raw experimental data, comprised of logs and configuration files (*see Appendix A*). Our analyses for this data fall into two categories that correspond to our two primary evaluation criteria: *low overhead* and *high granularity*.

5.1 Low overhead

The main idea behind low overhead is having a low, consistent impact on measured performance, which is important so that experimental instrumentation doesn't affect experiment results significantly (or inconsistently). The first way we examine this is through direct comparison of total system CPU utilization over the course of each experiment we ran. Table 2 shows the results of such an analysis, where we look at each experimental configuration that we have CPU utilization for (which includes every configuration that runs Collectl as part of its instrumentation). The main statistic of interest is average CPU utilization, measured in core-% (percentage of a single core), which is sampled in intervals of 1 second (otherwise, CPU utilization fluctuates significantly in small periods of time). This corresponds to 20 collection intervals per sampling interval at the target collection granularity of 50ms.

For a sampling interval c, the sampled CPU utilization at time $C(t)$ is calculated as follows:

$$C(t) = \frac{\Sigma_{k=i}^{j} x_k}{j - i + 1} \qquad \text{where } i, j \text{ are the min and max entry indices in } [t, t + c) \tag{1}$$

For a given configuration, we also divide each CPU utilization entry into two groups: "Load" and "Idle." To do this, we assign all CPU entries with utilization less than some factor k to be in "Idle," and all CPU entries with utilization greater than or equal to k to be in "Load." This factor is calculated **for each test run** as follows:

$$k = \frac{1}{2} \times Q_3 \qquad \text{where } Q_3 \text{ is the third quartile of CPU utilization values} \tag{2}$$

The most important statistic in the table is the percent difference from the control group's mean, which gives a general idea of *how much* overhead a tool has. The three values in the "Tool" category correspond to tests with Moby and Collectl, rAdvisor and Collectl, and just Collectl, each of which has CPU utilization information associated.

For the control group's mean μ_c and the mean of the group in interest, μ_k, the percent difference from the control group's mean, $\Delta_\mu\%$, is calculated as follows:

$$\Delta_\mu\% = \left(\frac{\mu_k}{\mu_c} - 1 \right) \times 100\% \tag{3}$$

Looking at the data, it is clear that rAdvisor has less direct CPU overhead than Moby during intervals considered to be in "Load," usually beating Moby by a large margin in all configurations but the 100% CPU tests. On the other hand, Moby seems to edge out rAdvisor by varying amounts during "Idle," though the significance of this is deceiving. For one, the percents by which Moby beats rAdvisor are inflated due to the small sizes of the numbers involved—the 4.34 core-% advantage Moby has over rAdvisor in the 50% `nbench` test at idle pales in comparison to the 26.36 core-% rAdvisor has over Moby in the same test during load. Additionally, the real-world significance of each category isn't the same: it's much more important to be able to measure system resource utilization with low, consistent overhead during actual system load than it is at

idle. Because of this, rAdvisor does much better in the direct CPU overhead measurement than Moby, exhibiting consistently low overhead across all experiments.

Table 2: Direct CPU overhead calculated for each applicable test configuration. The most important values have been highlighted blue for Moby and red for rAdvisor (for emphasis)

	Workload	Category	Tool	μ (core-%)	σ (core-%)	Δ_μ% to control	n
50 % CPU	`stress`	Load	Control	207.41	13.32	–	7,285
			Moby	236.11	15.24	+ **13.78** %	7,301
			rAdvisor	216.65	14.12	+ **4.45** %	7,292
		Idle	Control	14.50	18.90	–	3,214
			Moby	17.53	22.70	+ **20.96** %	3,270
			rAdvisor	18.22	18.77	+ **25.66** %	3,263
	`nbench`	Load	Control	205.64	11.30	–	15,561
			Moby	232.00	10.95	+ **12.82** %	15,505
			rAdvisor	214.69	11.42	+ **4.40** %	15,774
		Idle	Control	7.72	6.82	–	15,039
			Moby	7.43	7.60	− **3.76** %	15,261
			rAdvisor	11.77	7.73	+ **52.46** %	14,944
100 % CPU	`stress`	Load	Control	397.22	19.77	–	7,224
			Moby	397.43	18.25	+ **0.05** %	7,240
			rAdvisor	397.10	20.00	− **0.03** %	7,235
		Idle	Control	19.01	32.16	–	3,241
			Moby	21.40	34.05	+ **12.57** %	3,302
			rAdvisor	22.43	31.12	+ **17.99** %	3,304
	`nbench`	Load	Control	389.65	42.19	–	8,163
			Moby	392.79	33.47	+ **0.81** %	8,447
			rAdvisor	391.60	36.40	+ **0.50** %	8,222
		Idle	Control	7.92	11.01	–	22,417
			Moby	8.61	14.78	+ **8.71** %	22,235
			rAdvisor	12.41	12.60	+ **56.69** %	22,383
sustained	Microblog	Load	Control	23.34	24.66	–	41,693
			Moby	26.28	21.67	+ **12.60** %	48,203
			rAdvisor	23.72	23.01	+ **1.63** %	49,064
		Idle	Control	6.91	1.16	–	17,273
			Moby	7.45	1.79	+ **1.08** %	10,806
			rAdvisor	7.13	1.42	+ **3.18** %	9,916
bursty	Microblog	Load	Control	24.34	28.78	–	43,387
			Moby	27.78	25.41	+ **14.13** %	44,016
			rAdvisor	23.70	25.96	− **2.63** %	68,991
		Idle	Control	7.05	1.27	–	24,290
			Moby	8.00	2.41	+ **13.48** %	14,929
			rAdvisor	7.48	1.39	+ **6.10** %	19,215

5.1.1 Indirect overhead

In addition to the direct overhead comparison, we also obtained test-specific results that let us examine the *indirect* performance impact of the different tools. In these tests, we look at the performance impact of other running benchmarks on the same system, letting us know if the tools

can affect the scores of otherwise consistent benchmarks (and by how much).

The first of these benchmarks is `nbench`, which provides unit-less aggregate scores for memory, integer, and floating-point tests. In this analysis, we apply a similar methodology as the last one in that we find the control configuration's results (which corresponds to the configuration of just running Collectl) and then compare those to the results of additionally running other tools. For the scores of the configurations that include Moby and rAdvisor, we provide the delta-percentage between the experimental group and the control group, applying the same equation as before (3):

Table 3: Comparative results for different tool combinations using the `nbench` synthetic benchmark. Aggregate scores are unit-less.

	50 % peak CPU utilization			**100 % peak CPU utilization**		
Aggregate Score	**Control**	**Moby**	**rAdvisor**	**Control**	**Moby**	**rAdvisor**
memory	29.16	−0.72 %	−2.03 %	29.17	−0.78 %	−2.09 %
integer	26.49	−0.77 %	−1.92 %	26.54	−0.77 %	−1.56 %
floating point	41.54	−0.74 %	−2.00 %	41.62	−0.81 %	−1.65 %

Unlike the last result set, the results for `nbench` crown the patched version of Moby as a clear winner in having less indirect overhead, showing 1-2% less reduction in scores across the board. Though `nbench` is a synthetic benchmark, it seems like rAdvisor places more strain on system resources during its collection, causing increased resource contention and leaving less for the benchmark applications to consume. We provide possible explanations for this at the end (*see Conclusion*).

The last experiment type we tested uses the WISE microblog benchmark developed by the Elba project, which was adapted to operate in a containerized environment [30]. In it, we examine the end-to-end response time of HTTP API requests by intercepting Linux syscalls on the load generator machine, building a dataset of response times over time that we can then analyze.

In Table 4, we quantitatively look at the response time distribution for each tool/workload combination, which contains statistics for each quartile (Q_1, Q_2, and Q_3) as well as the 90^{th} percentile, all measured in milliseconds:

Table 4: Response time distributions for the microblog indirect overhead benchmark, including various percentiles and the number of collection entries used to develop the statistics from

		Percentiles				
Workload	**Tool**	Q_1 **(ms)**	Q_2 **(ms)**	Q_3 **(ms)**	$\mathbf{90^{th}}$ **(ms)**	n
bursty	Moby	34.68	57.91	101.24	219.50	10,836
	rAdvisor	33.64	56.43	98.78	206.62	10,820
	Neither	34.28	58.72	100.51	214.14	10,759
sustained	Moby	35.73	59.99	102.89	207.92	11,336
	rAdvisor	35.44	59.45	103.19	206.81	11,370
	Neither	35.76	59.04	101.02	207.11	11,292

We also take a visual look at the data by plotting response time histograms for each non-control tool/workload combination, which are displayed in Figure 17.

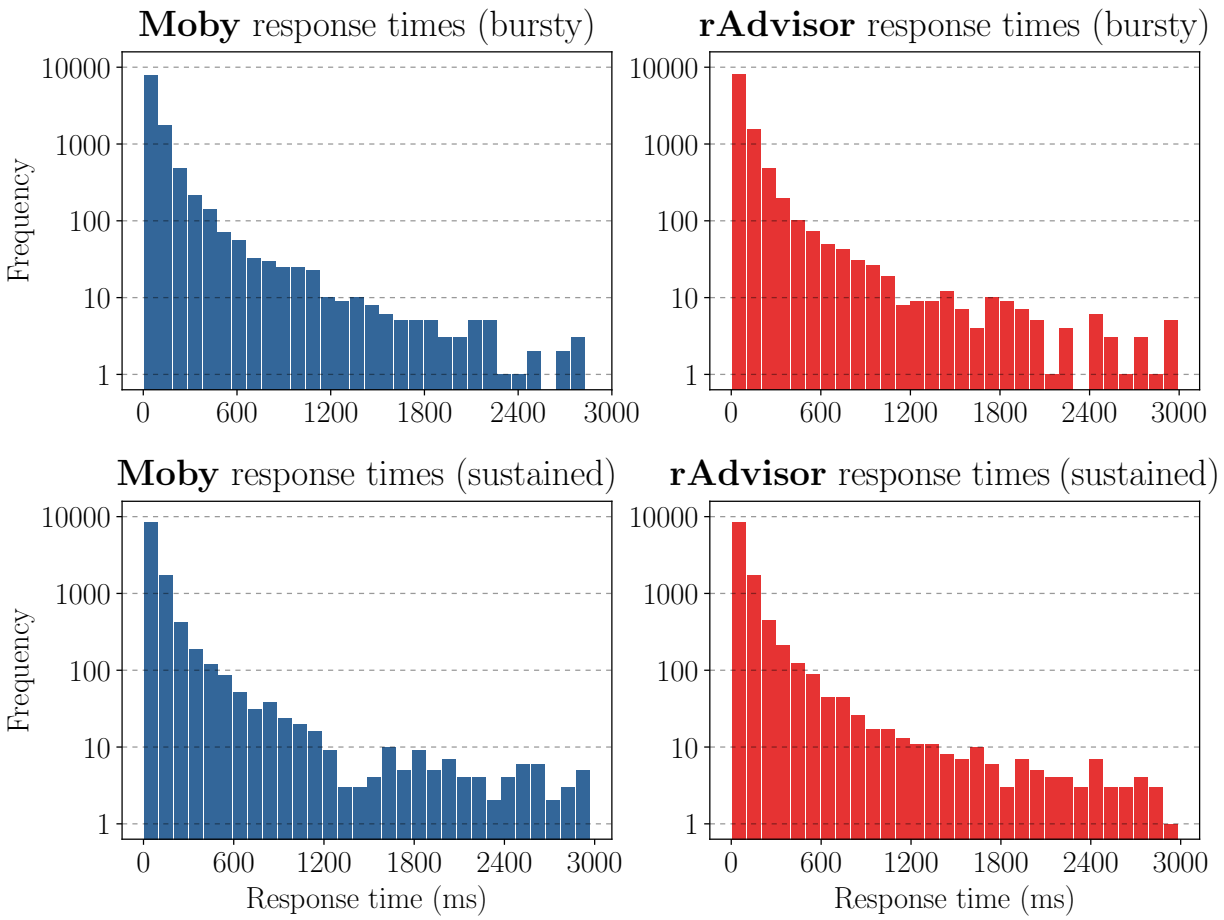

Figure 17: Response time histograms for both workload types (bursty and sustained) and both collection tools, with Moby data in blue and rAdvisor data in red

As the histograms hint at and the quantitative statistics show, there doesn't seem to be a significant difference in the response time distributions of the different test configurations. Having Moby exhibit seemingly *better* indirect overhead performance than the control group in some of the statistics, combined with the fact that the generated workloads are randomized, make the results of this test largely inconclusive, at least in examining the level of indirect overhead imposed by each tool.

Something the distributions *do* show, however, is the emergence of the long tail phenomenon in the containerized microblog benchmark, which will likely see more study in the next few years. Though almost all requests resolve in less than 200 milliseconds, a small fraction (less than 100 requests out of thousands) take longer than a second or two to resolve. This is expected, however, as the configuration used in this experiment is almost identical to the bare metal microblog configuration that has been shown to exhibit the long tail phenomenon in the past [10].

5.2 High granularity

The other evaluation criterion we are analyzing is whether the tools operate with high granularity—that is, whether they can collect with sufficiently small intervals, and do so consistently without degrading under load.

The first way we investigate this is by looking at all of the observed collection intervals across every experiment run we conducted, grouped together by which test/configuration they occurred during. This data is summarized in Tables 5 and 6, which include the observed collection intervals (in milliseconds) for each of the tools used (set to run at a 50 milliseconds target interval):

Table 5: Observed collection intervals for the direct `stress` and indirect `nbench` synthetic tests, which ran under either 50% or 100% CPU utilization limits (50ms target collection interval)

	Test	Tool	μ (ms)	σ (ms)	90%tile (ms)	Max (ms)	n
50% CPU	Direct	Moby	62.58	12.10	71.98	704.10	490,885
		rAdvisor	50.11	0.71	50.26	85.65	615,381
	Indirect	Moby	59.77	7.73	66.02	819.17	102,185
		rAdvisor	50.10	0.76	50.22	95.63	125,612
100% CPU	Direct	Moby	64.65	13.98	76.16	742.10	481,855
		rAdvisor	50.14	0.48	50.15	85.41	613,649
	Indirect	Moby	61.48	11.99	71.41	1275.67	52,883
		rAdvisor	50.11	0.52	50.13	97.04	59,493

Table 6: Observed collection intervals for the indirect microblog benchmark which ran using either a bursty or sustained representative workload (50ms target collection interval)

Workload	Tool	μ (ms)	σ (ms)	90%tile (ms)	Max (ms)	n
bursty	Moby	52.23	1.01	52.71	230.65	726,947
	rAdvisor	50.07	0.02	50.09	54.44	997,187
sustained	Moby	52.23	0.98	52.71	173.36	799,977
	rAdvisor	50.07	0.03	50.09	54.80	669,912

While the maximum value typically isn't particularly useful when analyzing distributions of data, it holds a specific significance in this problem, where even one regression in collection is unacceptable. This is because having significant regression in the collection interval (greater than 100 milliseconds) causes the assumptions made under the Sampling Theorem to break down, as you are no longer guaranteed to detect each discrete event that takes 100 milliseconds or more [12]. This is of particular importance in measuring millibottlenecks, where losing data over the course of a second, though seemingly short, is an eternity in the timescales being studied.

In addition to the tables, we also have created a set of scatterplots that visualize the relationship between container CPU utilization and observed collection interval for each experiment configuration/tool (Figure 18). The scatterplots graph observed intervals against rolling CPU averages, which are calculated using a rolling window size of 5. Like we discussed earlier, the reason to use some sort of averaging/sampling for CPU utilization statistics is because they're prone to rapid fluctuation, which can create misleading results. Additionally, each scatterplot shows only 20% of the data to reduce density, which may cause them to not display the global maximums.

Looking at the table and the scatterplots, it's clear that rAdvisor is able to achieve its target collection interval of 50 milliseconds significantly more consistently then Moby, which almost never attains the target interval. Additionally, as the standard deviations show, rAdvisor is very consistent with the intervals, showing very little variation compared to Moby. However, probably the most significant aspect of the data is whether there is an existence of a long tail. In the case of Moby, there is a significant long tail, with some observed collection intervals reaching into the thousands of milliseconds. As discussed above, missing an entire second of collection is unacceptable, so having this large of a long tail is a bad sign for Moby. rAdvisor, on the other hand, doesn't even go above 100 milliseconds for its collection interval over the 3 million collection events tested throughout all our experiment runs. This is a significant accomplishment, as it essentially shows the lack of existence of any long tail that would otherwise severely hurt the usefulness of the tool.

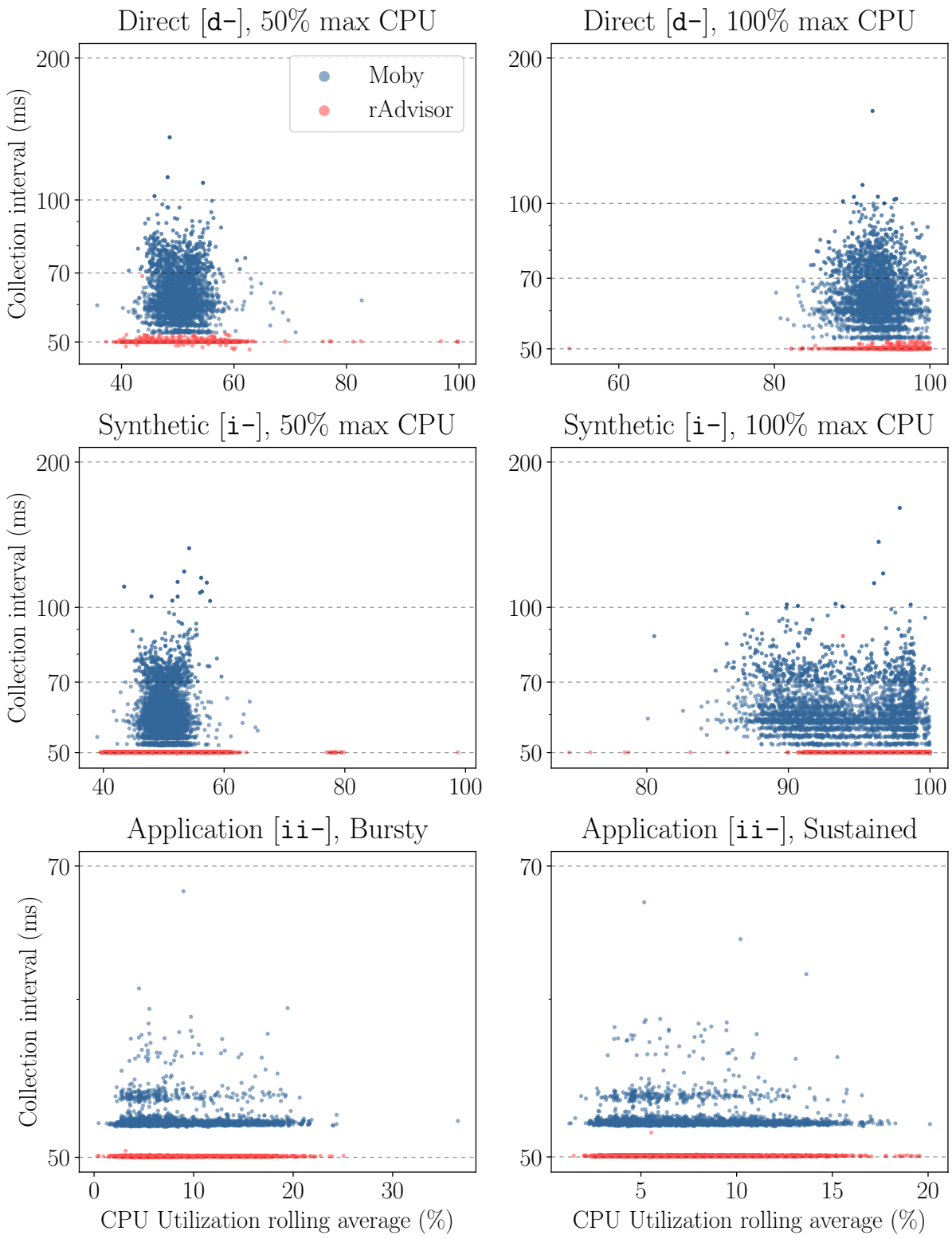

Figure 18: Set of scatterplots showing the relationship between CPU utilization and observed collection interval for both Moby (blue) and rAdvisor (red), with 20% of all data points shown

As an aside, there does appear to be some sort of banding in the scatterplots (especially in the top 4 Moby plots), though the cause of this is unknown. It's possible there was some number narrowing or rounding error during data analysis, though we're not exactly sure what the source is. It's also possible that the bands are an artifact of the timing mechanism used in the patched Moby version, which utilizes the Golang standard library's concurrency control features.

6 Conclusion

In our project, we've been able to investigate various system resource utilization monitors for Docker containers, eventually ending up developing our own custom solution in Rust. In order to be useful for examining the emergence of millibottlenecks in online systems, our tool needed to operate with *high granularity* and *low overhead*, the primary criteria we used to evaluate the tools. To examine the tools and compare their ability to meet these criteria, we performed 360 experiments in CloudLab, an experiment test bed for cloud research projects. Each experiment type tested a variety of different configurations and imposed a handful of both synthetic and application benchmarks. In the end, our custom-built tool, rAdvisor, provided several advantages over current tools such as being designed for the specific purpose of monitoring research systems (unlike current solutions that are built for real-world deployments [3], [4]). In direct CPU Utilization, rAdvisor had significantly less direct overhead during periods of load, and had little to no impact on the distribution of response times in application benchmarks. Moreover, rAdvisor was significantly more consistent at achieving its target collection interval than Moby across each test type. It never degraded past 100 milliseconds (+50 milliseconds from its target) in the 3 million collection events we observed, whereas Moby degraded as far as taking seconds between collection events, causing loss of all collected data during the periods of degradation. Though it did have an observable impact on the synthetic benchmark of the systems, we're confident we can reduce that in the future by modifying our underlying implementation (*see Future Work*).

6.1 Challenges

Throughout this project we faced many setbacks and challenges that caused us to pivot the scope of our project. At the beginning of the semester, our project proposal focused primarily on Kubernetes and we spent a lot of the first half of the project researching and implementing a Kubernetes cluster with the WISE microblog benchmark, yet ended up using none of it in our project. While researching monitoring tools that provided *high granularity* and *low overhead*, we also spent an excessive amount of time attempting to make cAdvisor, Sysdig, and Moby work to our standards, without avail. Having to change our project so far into the semester, after observing the lack of tool for monitoring that fit our needs, led to us having to complete our implementation very rapidly. To make this more challenging, we were unfamiliar with the tools and concepts involved, such as Rust, cgroups, and low-level programming. A few other challenges we faced were realizing we needed to create some sort of automation to complete our experiments on time (which we luckily were able to do) and having a large portion of experimental results wiped due to a hard drive failure.

6.2 Future Work

If the future, there are a lot of extensions and improvements to be made to the work we completed in this project. Our original purpose was to integrate Kubernetes with the WISE microblog benchmark to test for millibottlenecks, although we had to pivot our project halfway. Now that we have a tool, rAdvisor, that can monitor with *high granularity* and *low overhead* and supports Kubernetes pods, we can finish developing and running the Kubernetes experiment for the WISE microblog benchmark. We already made an experimental workflow script for CloudLab that creates an idle Kubernetes cluster on CloudLab, but with further work, we can investigate the emergence and propagation of millibottlenecks in containerized (and orchestrated) online systems.

There are also many different improvements that could be made to our experimental workflow. For one, we could add additional test types to test the relative overhead of rAdvisor and Moby to investigate if certain application domains are affected differently. We could look at more target collection intervals (other than 50 milliseconds), letting us investigate the limits of both tools and see if the target interval affects the overhead and observed granularity. Alternatively, though we used a microblog benchmark that only had certain parts containerized, we could fully containerize the application (including the web server frontend and database backend), letting us test the

collection tools on additional application types. At the end during analysis, we also had some difficulty in normalizing and comparing timestamps between log types—in the future, this could be a problem we tackle, using concurrency control mechanisms to guarantee synchronization between different hosts.

In rAdvisor itself, there are a number of improvements that could be made to improve its performance. As noted earlier, the synthetic benchmarks showed noticeable performance overhead, which we believe may be attributed to the design of the polling thread (which periodically polls the Docker daemon to get a list of active, running containers). There are a few expensive components to it, such as the bloated Docker API bindings library and the very expensive asynchronous executor that spawns multiple worker threads to execute HTTP requests. By replacing these with either a lightweight HTTP client to poll the Docker daemon or taking the shortcut of just listing the virtual directory for the Docker parent cgroup, we could eliminate a large portion of the performance overhead of the polling thread. Additionally, though we attempted to reduce the memory and CPU overhead of the collection thread, we didn't perform any significant profiling using CPU or memory tools such as Valgrind, which is a suite of tools for memory debugging, memory leak detection, and profiling [31]. By doing this, we could better optimize the runtime overhead of the program as a whole.

Additionally, we could make direct improvements to the versatility of rAdvisor by implementing additional features. For example, rAdvisor currently lacks the functionality to instrument network statistics for containers, compared to Moby, which does. The reasoning for this is because networking isolation in Linux and Docker doesn't use cgroups, and instead uses entirely different mechanisms (namely, network namespaces) that would require significant additional complexity [19]. According to the Docker documentation, instrumentation software would have to fork a new process per container, switch the network namespace for each, and collect network statistics in those processes. From here, we would also have to implement IPC and child process control, which we deemed to be outside of the scope of the current project. Another possible feature to add would be adapting rAdvisor to also work on Windows systems. Since 2016, Microsoft has made available a set of high level APIs called Host Compute Services, used to instantiate and manage first-party Windows containers in the kernel [32] (which is the foundation of Windows functionality for Docker). With this, Docker containers can be created that run Windows Server, a niche workflow common in fields such as .NET development. Using the HCS API, we could add Windows support for system resource utilization instrumentation to rAdvisor, which already compiles on Windows due to Rust's excellent cross platform support.

6.3 Final Words

Throughout the course of this project, we explored the innerworkings of collection tools such as Sysdig, cAdvisor, Collectl, and Moby. We soon realized the flaws and saw the need for a tool that could prioritize high granularity and low overhead while still delivering consistent results. rAdvisor delivers on all of our goals, and contains many efficiencies necessary for research needs. Even in the areas that rAdvisor underperforms in, we have a clear path forward to improve its performance and better optimize it. We hope that rAdvisor can serve the purpose of a useful collection tool, and eventually be used to detect millibottlenecks in containerized and orchestrated systems.

References

[1] (Apr. 2020). Docker overview, [Online]. Available: https://docs.docker.com/engine/docker-overview/ (visited on 04/23/2020).

[2] Portworx and A. Security, *2019 container adoption survey*, May 2019. [Online]. Available: https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf (visited on 04/22/2020).

[3] SysDig, *Sysdig*, https://github.com/draios/sysdig, GitHub repository, 2014.

[4] Google, *Cadvisor*, https://github.com/google/cadvisor, GitHub repository, 2014.

[5] C. Pu, J. Kimball, C.-A. Lai, T. Zhu, J. Li, J. Park, Q. Wang, D. Jayasinghe, P. Xiong, S. Malkowski, Q. Wu, G. Jung, Y. Koh, and G. Swint, "The millibottleneck theory of performance bugs, and its experimental verification," *IEEE*, vol. 37, no. 10, pp. 1919–1926, 2017. DOI: 10.1109/ICDCS.2017.198.

[6] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, 2 Feb. 2013. DOI: 10.1145/2408776.2408794.

[7] T. Hoff, "Google: Taming the long latency tail - when more machines equals worse results," [Online]. Available: http://highscalability.com/blog/2012/3/12/google-taming-the-long-latency-tail-when-more-machines-equal.html (visited on 04/23/2020).

[8] R. Lima, J. Kimball, J. E. Ferreira, and C. Pu, "Systematic construction, execution, and reproduction of complex performance benchmarks," D. Da Silva, Q. Wang, and L.-J. Zhang, Eds., pp. 26–37, 2019. DOI: 10.1007/978-3-030-23502-4_3.

[9] Collectl, [Online]. Available: http://collectl.sourceforge.net/ (visited on 04/23/2020).

[10] R. Lima. Experimental verification of the millibottleneck theory of performance bugs with the wise toolkit, [Online]. Available: https://www.cc.gatech.edu/~ral3/tutorial2.html (visited on 04/27/2020).

[11] What is kubernetes? [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[12] E. Lai, "2 - converting analog to digital signals and vice versa," in *Practical Digital Signal Processing*, E. Lai, Ed., Oxford: Newnes, 2003, pp. 14–49, ISBN: 978-0-7506-5798-3. DOI: 10.1016/B978-075065798-3/50002-3. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780750657983500023.

[13] SysDig, *Sysdig*, https://github.com/draios/sysdig/blob/a259b4bf49c3330d9ad6c3eed9eb1a31954259a6/userspace/sysdig/csysdig.cpp#L435-L438, GitHub repository, 2014.

[14] C. Doxsey, "Golang book," [Online]. Available: https://www.golang-book.com/books/intro/10 (visited on 04/24/2020).

[15] R. Lima, *Wisetrace*, https://github.com/coc-gatech-newelba/WISETutorial/blob/60cd8f481afdd6773b6148f88dc1ba19fe4498e5/WISETrace/kernel_modules/recvfrom/spec_recvfrom.c#L23-L34, GitHub repository, 2018.

[16] S. Klabnik and C. Nichols, "The rust programming language," [Online]. Available: https://doc.rust-lang.org/book/title-page.html (visited on 04/24/2020).

[17] M. Kerrisk. (2008). Linux control groups, [Online]. Available: http://man7.org/linux/man-pages/man7/cgroups.7.html (visited on 04/25/2020).

[18] P. Menage. Cgroups, [Online]. Available: https://kernel.org/doc/Documentation/cgroup-v1/cgroups.txt (visited on 04/25/2020).

[19] Runtime metrics, [Online]. Available: https://docs.docker.com/config/containers/runmetrics/ (visited on 04/25/2020).

[20] Get data useage information, [Online]. Available: https://docs.docker.com/engine/api/v1.30/#operation/ServiceUpdate (visited on 04/25/2020).

[21] Csvy: Yaml frontmatter for csv file format, [Online]. Available: https://csvy.org/ (visited on 04/26/2020).

[22] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19.

[23] "Cloudlab technology," [Online]. Available: https://www.cloudlab.us/technology.php (visited on 04/29/2020).

[24] Stress, [Online]. Available: http://web.archive.org/web/20190707055144/http://people.seas.harvard.edu:80/~apw/stress/ (visited on 04/26/2020).

[25] "Bytemark," 2011. [Online]. Available: http://web.archive.org/web/20160304001428/https://www.tux.org/~mayer/linux/byte/bdoc.pdf (visited on 04/26/2020).

[26] U. Mayer. Linux/unix nbench, [Online]. Available: https://www.math.utah.edu/~mayer/linux/bmark.html (visited on 04/26/2020).

[27] The selenium browser automation project, [Online]. Available: https://www.selenium.dev/documentation/ (visited on 04/26/2020).

[28] N. Spurrier. Pxssh - control an ssh session, [Online]. Available: https://pexpect.readthedocs.io/en/stable/api/pxssh.html (visited on 04/27/2020).

[29] T. Khotiaintseva, "Retrying after exceptions, and handling internet connection problems," [Online]. Available: https://pragmaticcoders.com/blog/retrying-exceptions-handling-internet-connection-problems/ (visited on 04/27/2020).

[30] D. P. Neelappa, *Elbacontainer*, https://github.com/daniel-neelappa/elbacontainer, GitHub repository, 2020.

[31] Valgrind: About, [Online]. Available: https://valgrind.org/info/about.html.

[32] S. Cooley and I. Ashimine, "Introducing the host compute service (hcs)," 2020. [Online]. Available: https://docs.microsoft.com/en-us/virtualization/community/team-blog/2017/20170127-introducing-the-host-compute-service-hcs.

A Artifacts

During the completion of our project, we produced a variety of data, code, and documentation artifacts, which are listed below:

A.1 GitHub repositories

- elba-docker/radvisor - repository for rAdvisor, the primary artifact we created for the project, which is our custom Rust-based Docker container stats collection tool.
- elba-docker/moby (fork of moby/moby) - repository for our patched version of Moby, another primary artifact of the project, where we introduce configurable automatic statistics collection for running containers.
- elba-docker/experiment (fork of coc-gatech-newelba/WISETutorial) - contains the experimental workflow Bash scripts, configuration files, and Dockerfiles for each of the three test types, which run the experiments in CloudLab cluster. Also, the repository contains the (minor) modifications made to the microblog benchmark, which are mostly related to the differences in database user inference between bare metal and containerized environments (since containers run as the `root` user by default).
- elba-docker/automation - contains the Python automation scripts used to automate provisioning and running experiments on CloudLab, utilizing Selenium to automate creating and terminating experiments according to a YAML configuration file.
- elba-docker/analysis - collection of Python scripts and Jupyter Notebooks used for the parsing and analysis of our generated experimental data (used to create the graphs/tables in this report).
- elba-docker/instrumentation - working area used over the process of evaluating various system instrumentation for Docker, containing benchmarks, log files, and initial parsers/analysis notebooks for Moby, cAdvisor, Sysdig, and rAdvisor.
- elba-docker/sysdig (fork of draios/sysdig) - contains our minor modification to the upstream Sysdig repository where we remove the 100 millisecond lower bound on the collection interval
- elba-docker/nbench (fork of zosxang/nbench) - contains our pre-compiled Linux-amd64 binary under "Releases" that we use for running nbench in our experiments.
- elba-docker/services (fork of daniel-neelappa/elbacontainer) - contains the modifications we made to the Dockerfiles for the Elba microblog benchmark application, which mostly consist of adding the ability to pass in a database user to the contained microservices.

A.2 Docker containers

First, there are the five microservice containers built from the modified Dockerfiles in elba-docker/services, which reflect the additional capability to include the database username when running:

- wisebenchmark/auth (Source Dockerfile)
- wisebenchmark/inbox (Source Dockerfile)
- wisebenchmark/queue (Source Dockerfile)
- wisebenchmark/blog (Source Dockerfile)
- wisebenchmark/sub (Source Dockerfile)

There are also the Docker images used to bootstrap the benchmark containers in elba-docker/experiment:

- jazevedo6/indirect_throughput (Source Dockerfile) - used to execute the direct CPU overhead test by pre-installing `stress` on an Ubuntu base image
- jazevedo6/direct_collectl (Source Dockerfile) - used to execute the indirect synthetic benchmark test by pre-installing `nbench` on an Ubuntu base image

A.3 Data

The final data for the three different experiments types is available on Google Drive at:
drive.google.com/open?id=1I8FN0b_y37_vpGmf2XM0oL0W-5GsHbj0

A.4 Compiled binaries

- Patched moby Debian (`.deb`) packages - we used the existing tools at docker/docker-ce to build Debian packages for our custom version of Moby, which we then install during experiment execution on the Ubuntu bare metal hosts.
- rAdvisor binaries - we use GitHub actions to automatically build and distribute binaries of rAdvisor upon release, which we then directly download and run during experiment execution.

A.5 Supplemental documentation

During development, we wrote detailed documentation on each column in the rAdvisor output logs, documenting which cgroup file they come from and what they mean. The documentation is available here: elba-docker/radvisor/docs/collecting.md

We also wrote documentation on the command-line usage of rAdvisor, which is available here: elba-docker/radvisor/man/