Rensselaer

Submitty

Rensselaer Center for Open Source

Comparing Jailed Sandboxes vs Containers Within an Autograding System

Matthew Peveler
Evan Maicus
Barbara Cutler

SCAN ME

RCOS
Rensselaer Center for Open Source

https://submitty.org/

Submitty

Rensselaer Center for Open Source

- A free, open source autograding platform.
  - ~2500 users
  - 12-15 courses supported per term at RPI
  - In operation since 2014
- Support for:
  - Assignment Submission
  - Autograding
  - Exam Grading/Scanned PDF upload
  - Course Communications (Email/Forum)
  - Course Material Hosting
  - Plagiarism Detection
- https://submitty.org

What is Autograding?

- Student submits their code (either through direct upload or VCS) to a server
- Code is then compiled (if necessary) and executed and then run against some number of testcases determined by instructor
- The result of the testcases (generally program output) is then compared against expected input provided by instructor and graded
- Students can resubmit to improve their grades if they were incorrect on some testcases

Autograding System Concerns

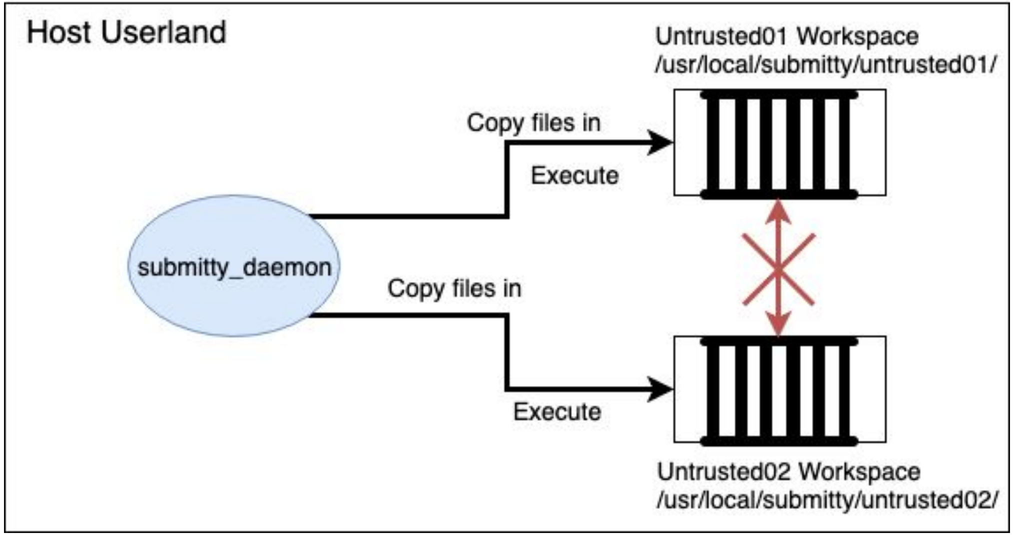
- Students are autograded in parallel with other students
- Prevent student from accessing other student’s code (or hidden instructor input)
- Student should not be able to affect the host system’s files and processes
- Student should not be able to take up an excessive amount of resources (CPU, RAM, etc.)
- Run code in many different languages and versions and their dependencies
    - Python and Pip
    - Ruby and Bundler
    - C++ and Boost/libev/etc

Talk Outline

- Grading via Jailed Sandbox
  - Autograding Workflow
  - Handling Security and Resources
- Grading via Docker
  - Handling different languages and dependencies
- Performance Analysis
  - Comparison stress test of two approaches

What is a Jailed Sandbox?

- Method of untrusted execution on the "bare metal" using installed global programs
- Submissions are graded in parallel
- Execute student code as "untrusted" user with limited file access



Sandbox Autograding Workflow

1. Student submits their code
2. Copy the student’s code into the untrusted work area
3. Copy instructor files for testcase into work area
4. (If needed) Compile the student’s code as untrusted
5. Run the student’s code as untrusted
6. Validate results
7. Move final results into a final directory outside of untrusted work area
8. Delete all files in untrusted work area

Handling Security

- Use rlimits on CPU, RAM, number of open files, maximum file size, etc. to prevent students from doing anything excessive

- Use seccomp to prevent students from accessing system calls unnecessary for grading (like fork in C)

- File permissions to limit student’s code from accessing anything at large in the file system

Executing User’s Code

- Run as untrusted user, but uses programs from /bin, /usr/bin, /usr/local/bin, etc.
- To support multiple languages/versions, just need to have executable in one of those locations
    - Python3.4 -> /usr/local/bin/python3.4
    - Python3.5 -> /usr/local/bin/python3.5

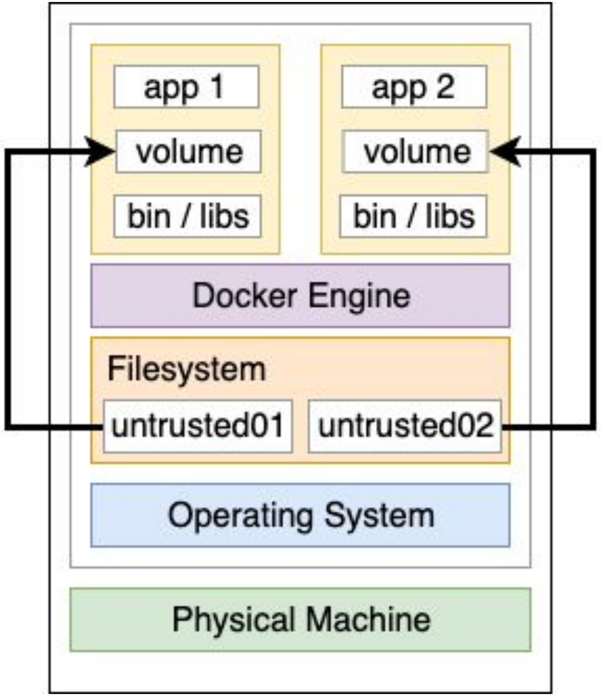
- Dependencies for those languages are installed at global scope so all untrusted users have access to them
    - pip3.4 install numpy
    - pip3.5 install numpy

Handling Differing Dependencies of Languages

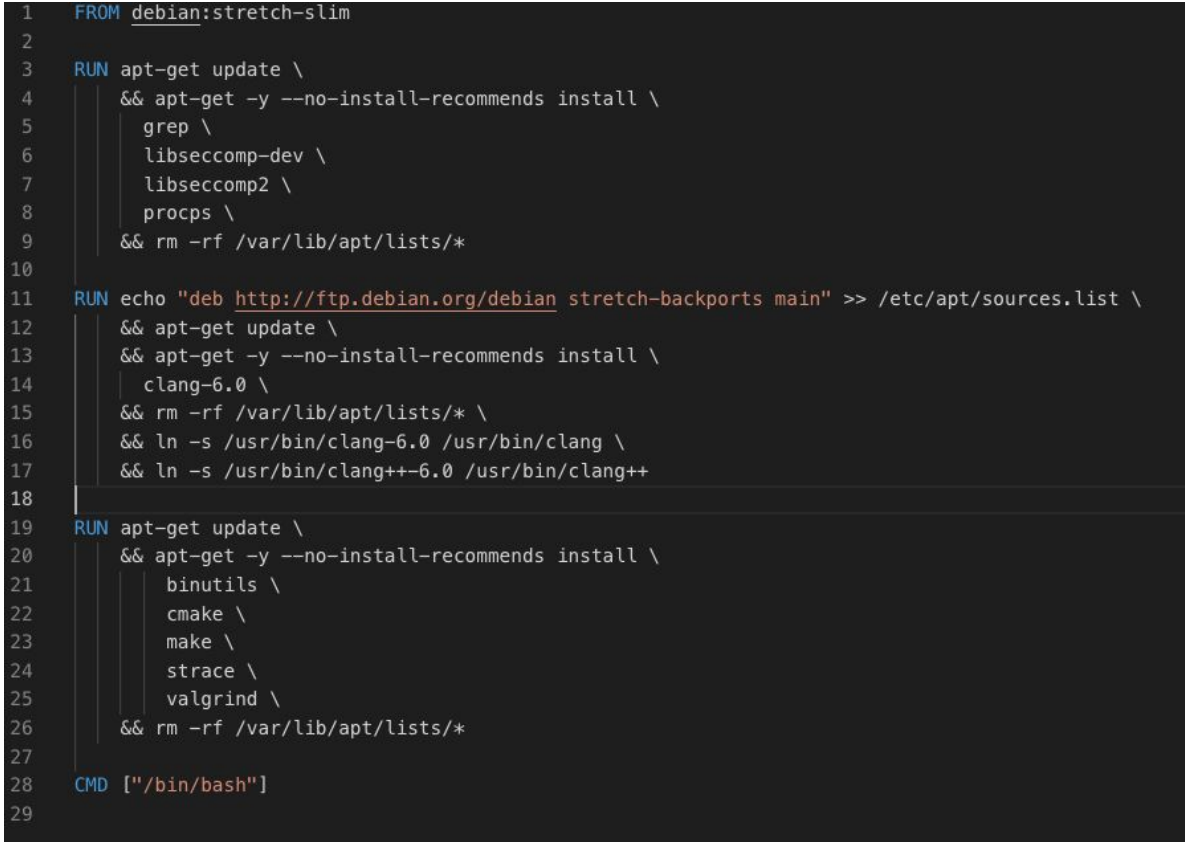
- Two instructors want to both use Python 3.5
- Instructor 1 wants to use numpy, Instructor 2 does not
- Options:
  - Install two versions of Python3.5
  - Spin up virtualenvs for autograding
  - Copy dependencies into sandbox and set custom PYTHONPATH
- How to handle similar situations for other languages with global dependencies?
  - C++: boost (apt-get install boost)
  - Ruby: bundler
- Increased workload on sysadmins
- How to know when certain packages are no longer used/needed
- Solution is complicated and language specific to support

What is Docker?

- Program that performs OS-level virtualization ("containerization")
- Shares resources with the host kernel and userland and so not as heavy-weight as a full virtual machine (which is fully virtualized)
- Resources within the container are still isolated from host
- Containers are created for a user created "Dockerfile" / image
- Can share "volumes" (folders) from host system to a container during runtime



What is a Dockerfile / image



```
FROM debian:stretch-slim

RUN apt-get update \
    && apt-get -y --no-install-recommends install \
      grep \
      libseccomp-dev \
      libseccomp2 \
      procps \
    && rm -rf /var/lib/apt/lists/*

RUN echo "deb http://ftp.debian.org/debian stretch-backports main" >> /etc/apt/sources.list \
    && apt-get update \
    && apt-get -y --no-install-recommends install \
      clang-6.0 \
    && rm -rf /var/lib/apt/lists/* \
    && ln -s /usr/bin/clang-6.0 /usr/bin/clang \
    && ln -s /usr/bin/clang++-6.0 /usr/bin/clang++

RUN apt-get update \
    && apt-get -y --no-install-recommends install \
        binutils \
        cmake \
        make \
        strace \
        valgrind \
    && rm -rf /var/lib/apt/lists/*

CMD ["/bin/bash"]
```

- Image gets built once
- As many containers as you want get spin up from built image
- Keep images small and free of unnecessary bloat

Docker Autograding Workflow

1. Student submits their code
2. Copy the student’s code into the untrusted work area
3. Copy instructor files for testcase into work area
4. **Create Docker Container mounting untrusted work area as volume**
5. (If needed) Compile the student’s code as untrusted **in container**
6. Run the student’s code as untrusted **in container**
7. Validate results
8. Move final results into a final directory outside of untrusted work area
9. **Stop and destroy any running containers**
10. Delete all files in untrusted work area

Handling Security

- Similar to Sandbox (rlimit, seccomp, etc.)
- Set limits on container itself, but this has problems with some languages
  - E.g. Java 8 and earlier sees host properties, not container limits (https://royvanrijn.com/blog/2018/05/java-and-docker-memory-limits/)
- Docker runs everything by default as root, make sure to set different user when executing commands
- The user running Docker must be trusted and treated as a sudo user

Handling Differing Dependencies of Language

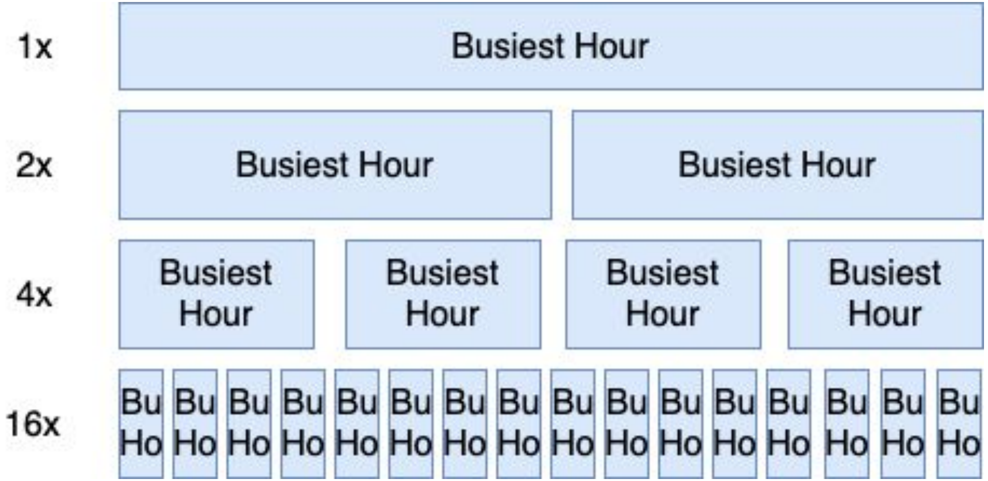
- Can create any number of images for any configuration of languages and dependencies
- Image 1:
  - Python 3.5
  - Numpy
  - Clang-6
  - Boost
- Image 2:
  - Python 3.5
- No conflict or special configuration necessary beyond creating custom Dockerfiles / images

Performance Analysis

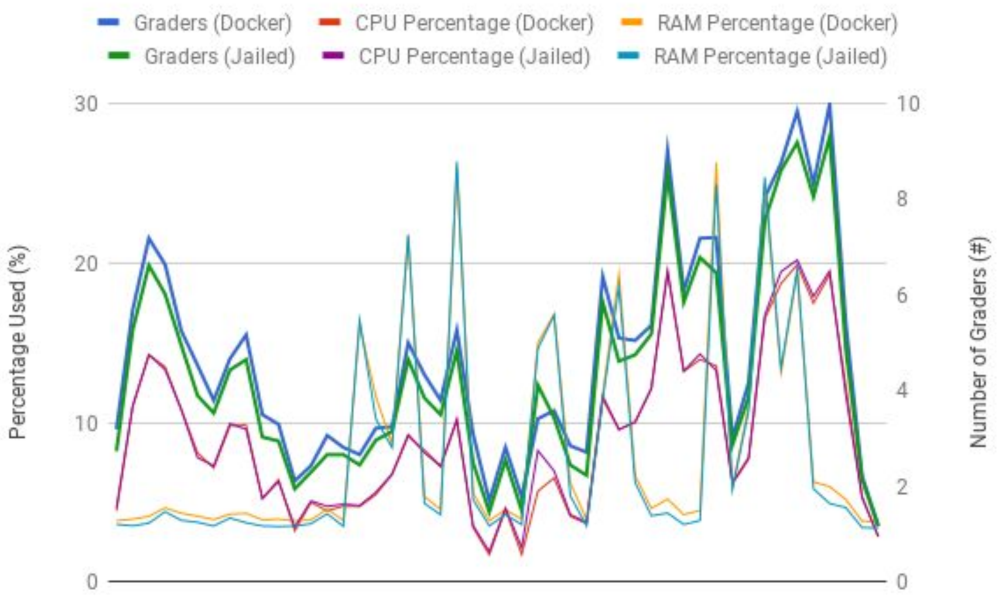
- What is the cost of running autograding in sandbox vs docker?
- Measure:
    - Time taken per student
        - Time spent waiting for process
        - Time spent grading
    - Resources used over time by the system
        - CPU
        - RAM

Experiment Setup

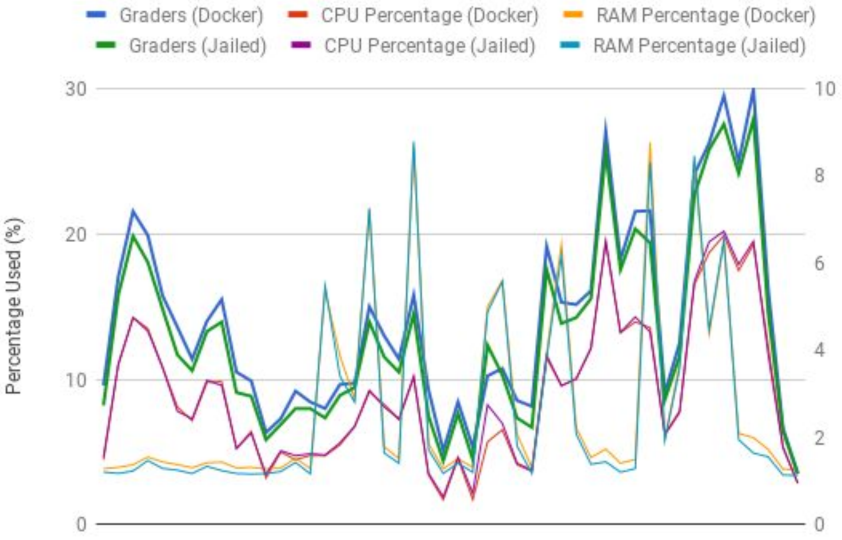
- Take busiest hour slice from mid-Fall 2017 and replay it at various speeds
- ~540 submissions from CS1 and 2
- Measure every second:
  - CPU
  - RAM
  - Active Graders
- For each submission, measure:
  - Time entered wait queue for grader
  - Time submission started graded
  - (if Docker) time to start container
  - Time grading finished
  - (if Docker) time to kill container

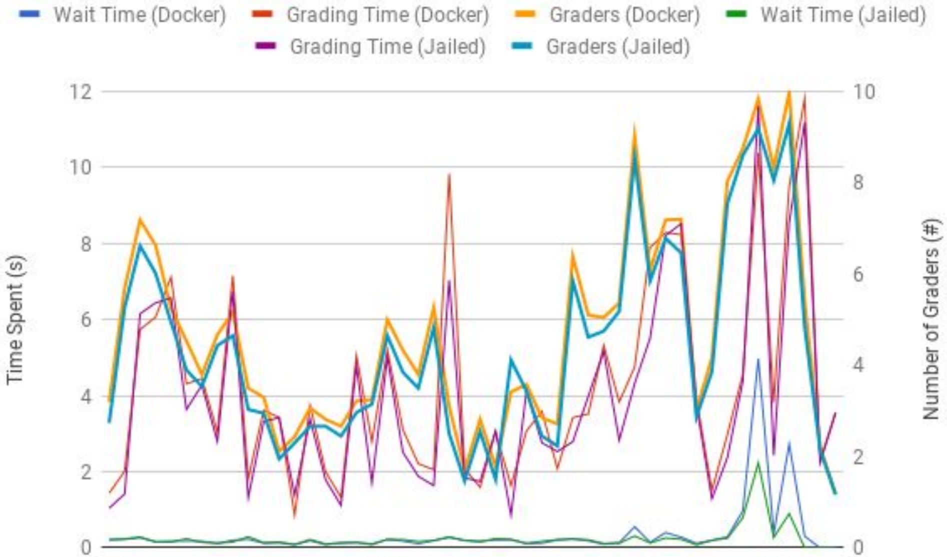


Results (4x)

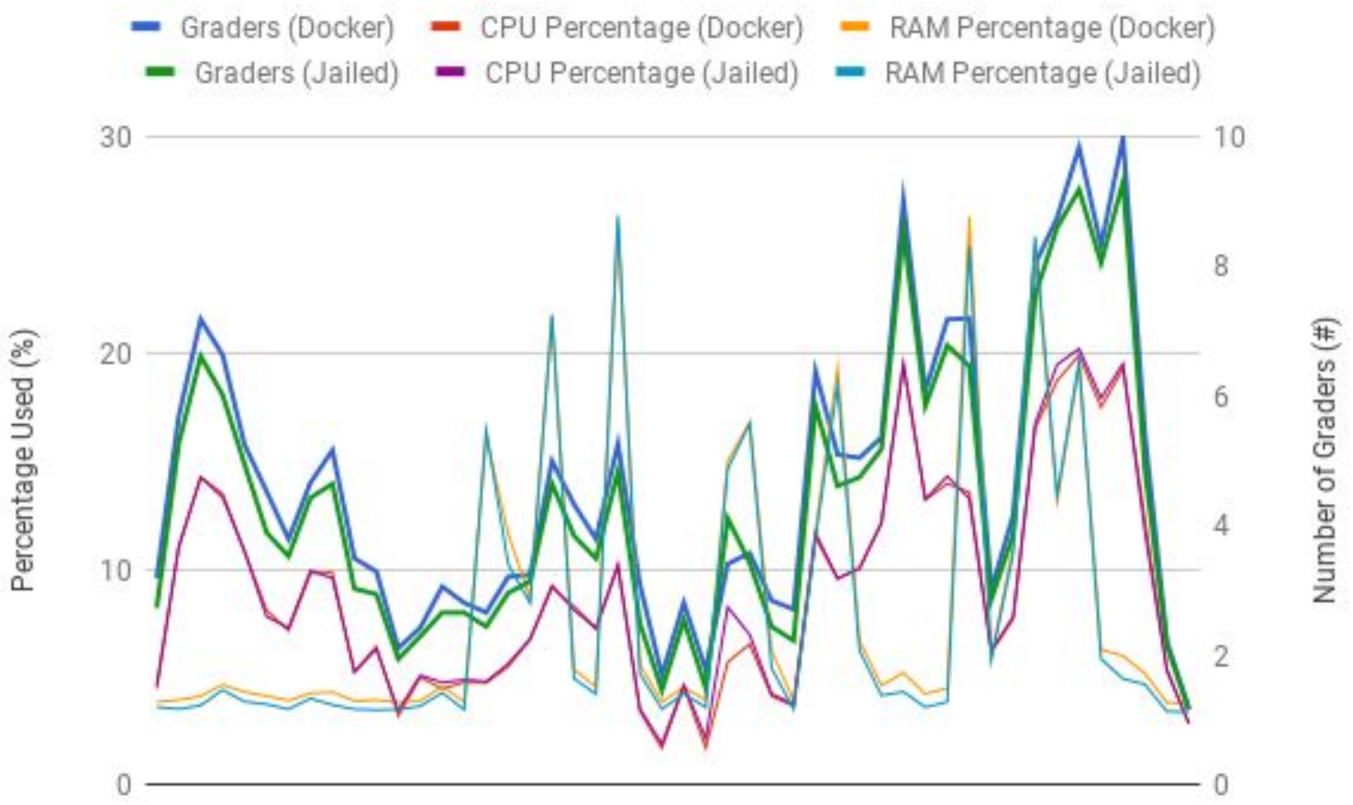

Graders (Docker)
CPU Percentage (Docker)
RAM Percentage (Docker)
Graders (Jailed)
CPU Percentage (Jailed)
RAM Percentage (Jailed)
Percentage Used (%)
30
20
10
0
Number of Graders (#)
10
8
6
4
2
0

Results (4x)

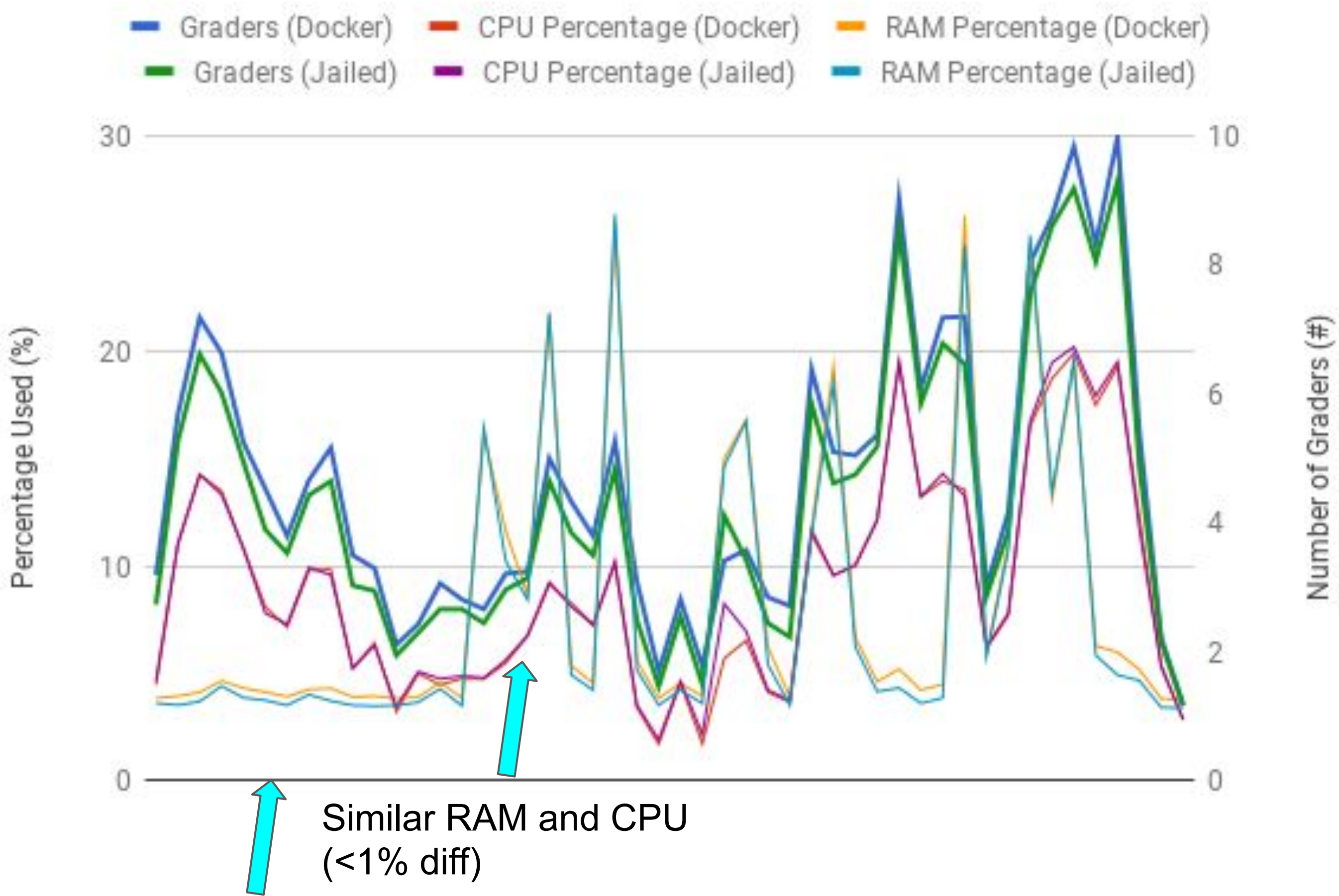

Graders (Docker)
CPU Percentage (Docker)
RAM Percentage (Docker)
Graders (Jailed)
CPU Percentage (Jailed)
RAM Percentage (Jailed)
Percentage Used (%)
30
20
10
0
Number of Graders (#)
10
8
6
4
2
0


Wait Time (Docker)
Grading Time (Docker)
Graders (Docker)
Wait Time (Jailed)
Grading Time (Jailed)
Graders (Jailed)
Time Spent (s)
12
10
8
6
4
2
0
Number of Graders (#)
10
8
6
4
2
0

Results (4x) - Resource Usage

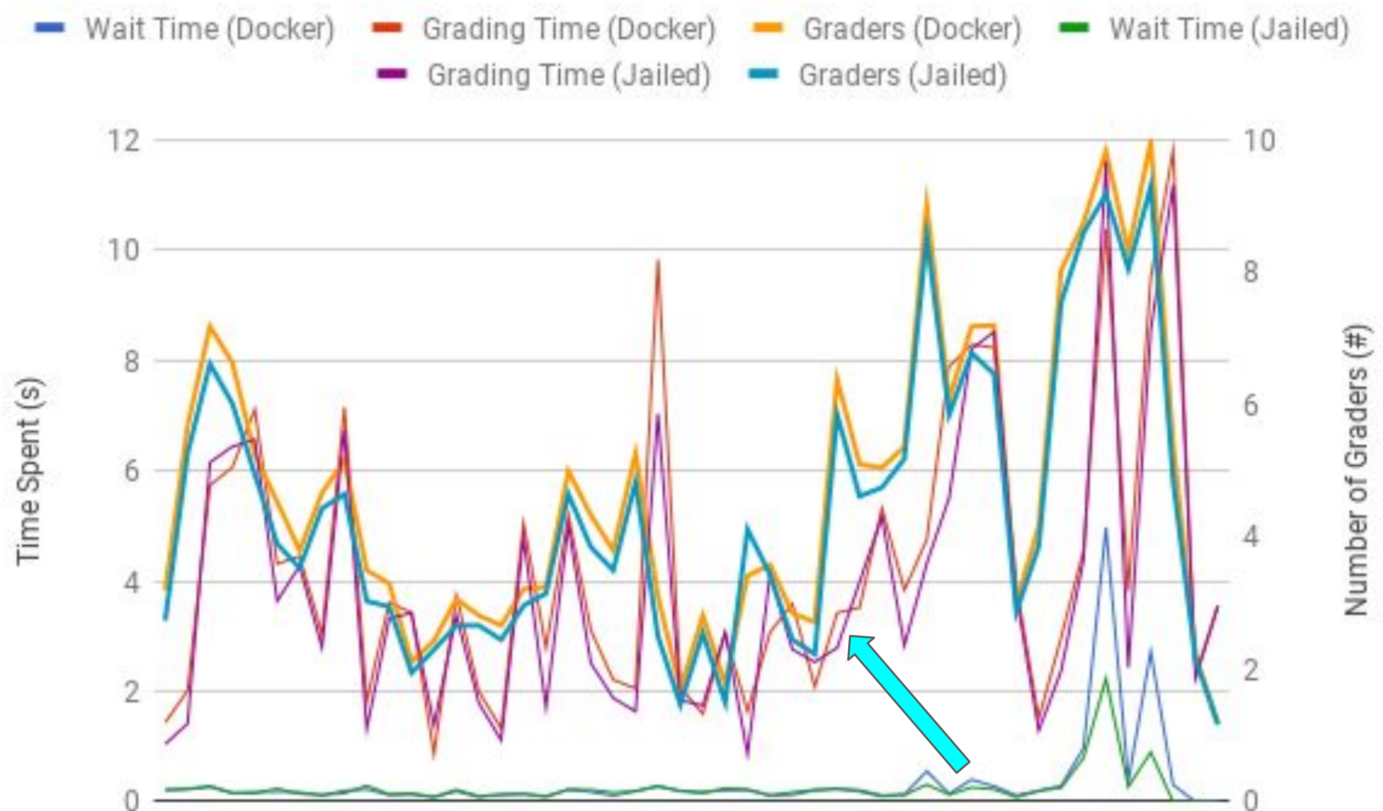


Results (4x) - Resource Usage



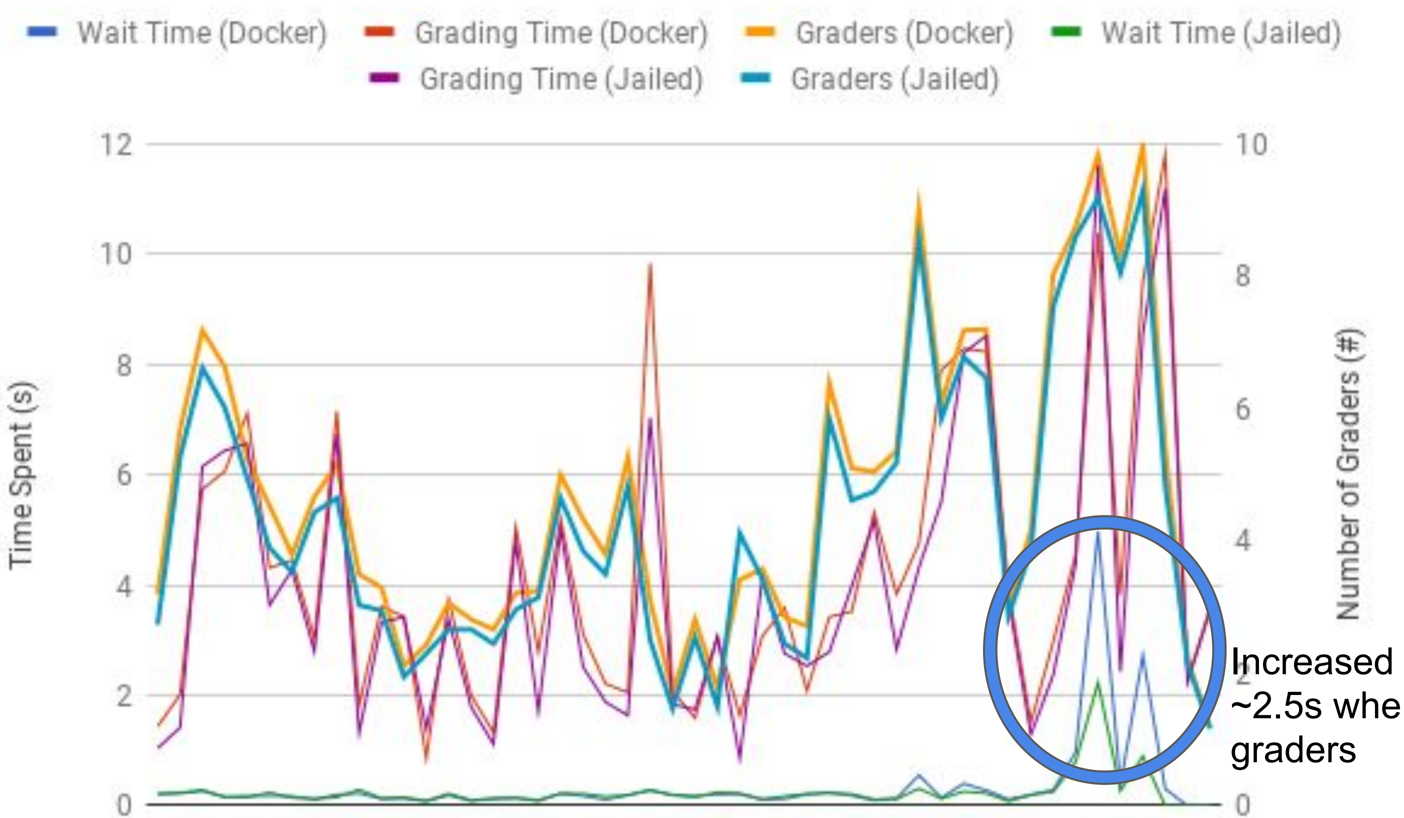
Similar RAM and CPU
(<1% diff)

Results (4x) - Time Spent



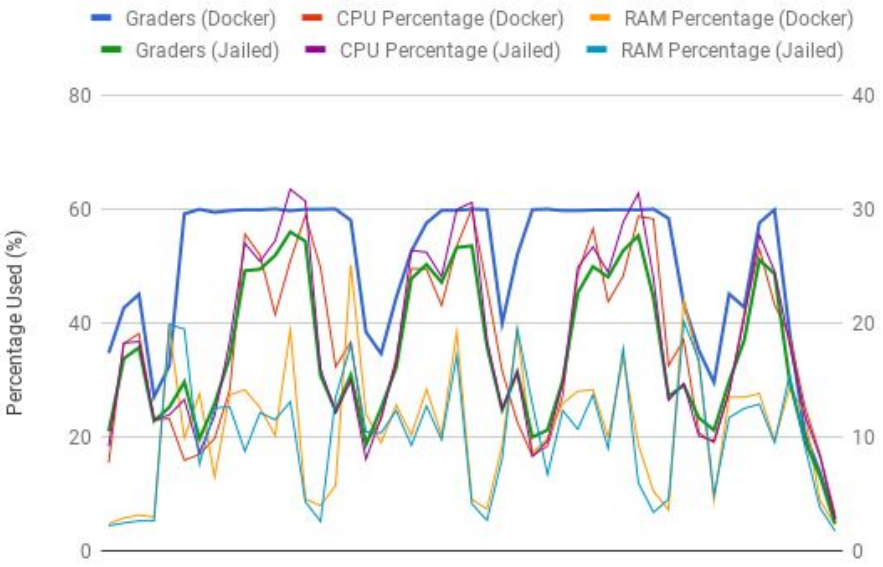
Similar wait times (0s) and grade times (~5s)

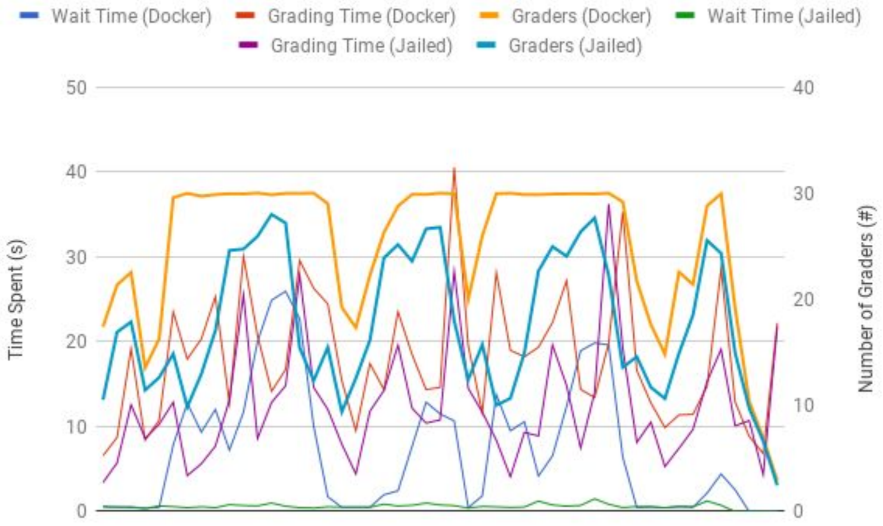
Results (4x) - Time Spent



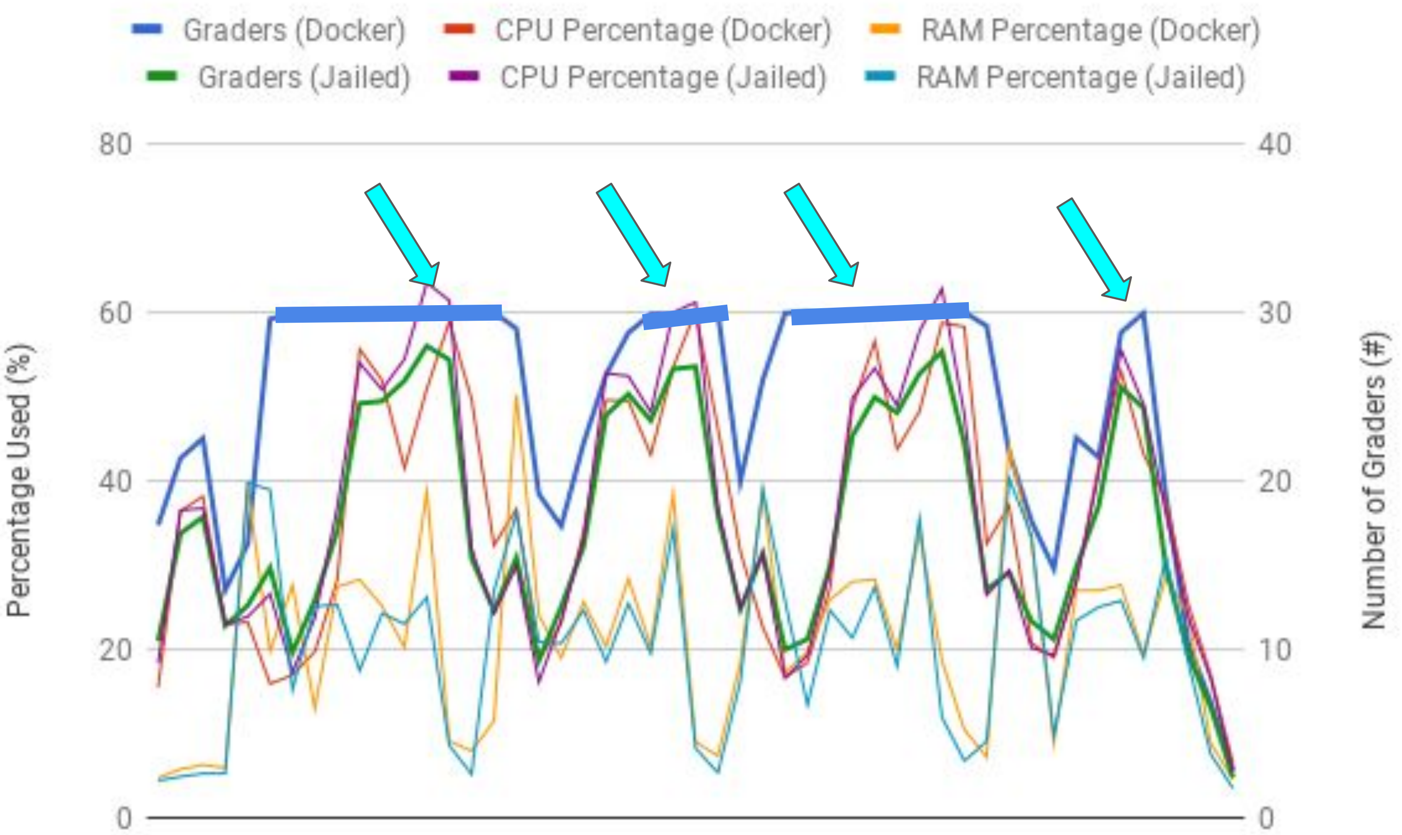
Increased wait time of ~2.5s when hit max graders

Results (16x)

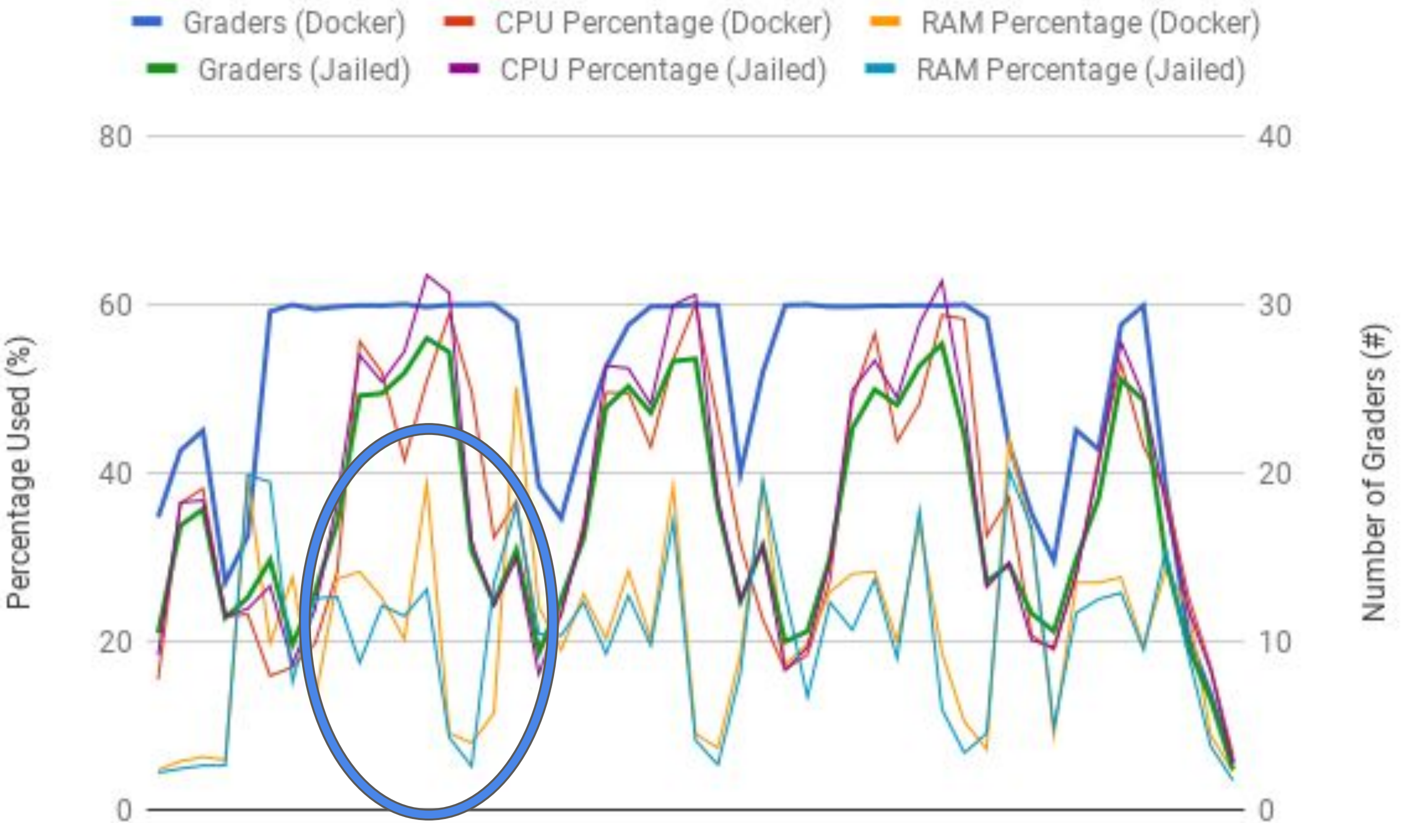

Graders (Docker)
CPU Percentage (Docker)
RAM Percentage (Docker)
Graders (Jailed)
CPU Percentage (Jailed)
RAM Percentage (Jailed)
Percentage Used (%)
Number of Graders (#)
0
20
40
60
80
0
10
20
30
40


Wait Time (Docker)
Grading Time (Docker)
Graders (Docker)
Wait Time (Jailed)
Grading Time (Jailed)
Graders (Jailed)
Time Spent (s)
Number of Graders (#)
0
10
20
30
40
50
0
10
20
30
40

Results (16x) - Resource Usage

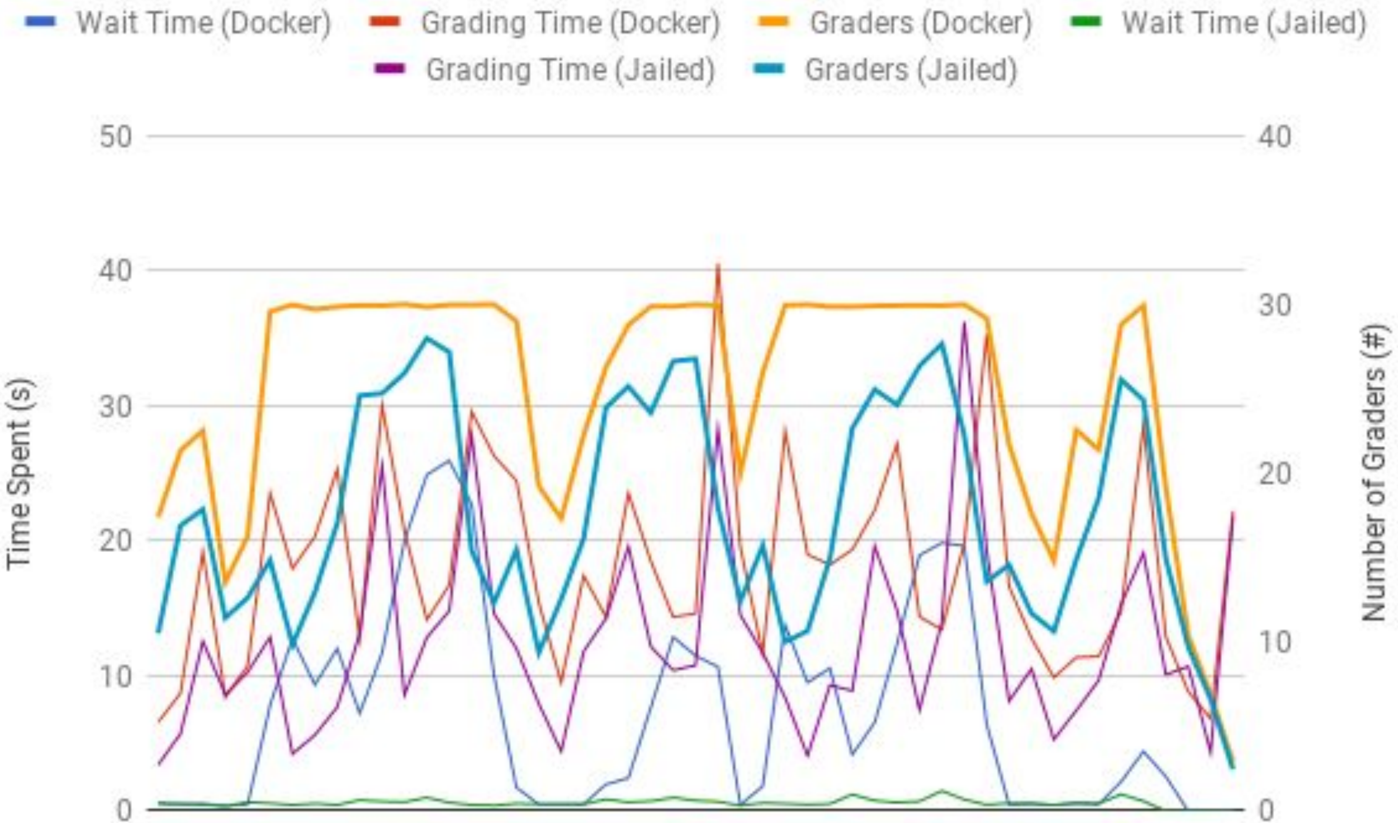


Results (16x) - Resource Usage

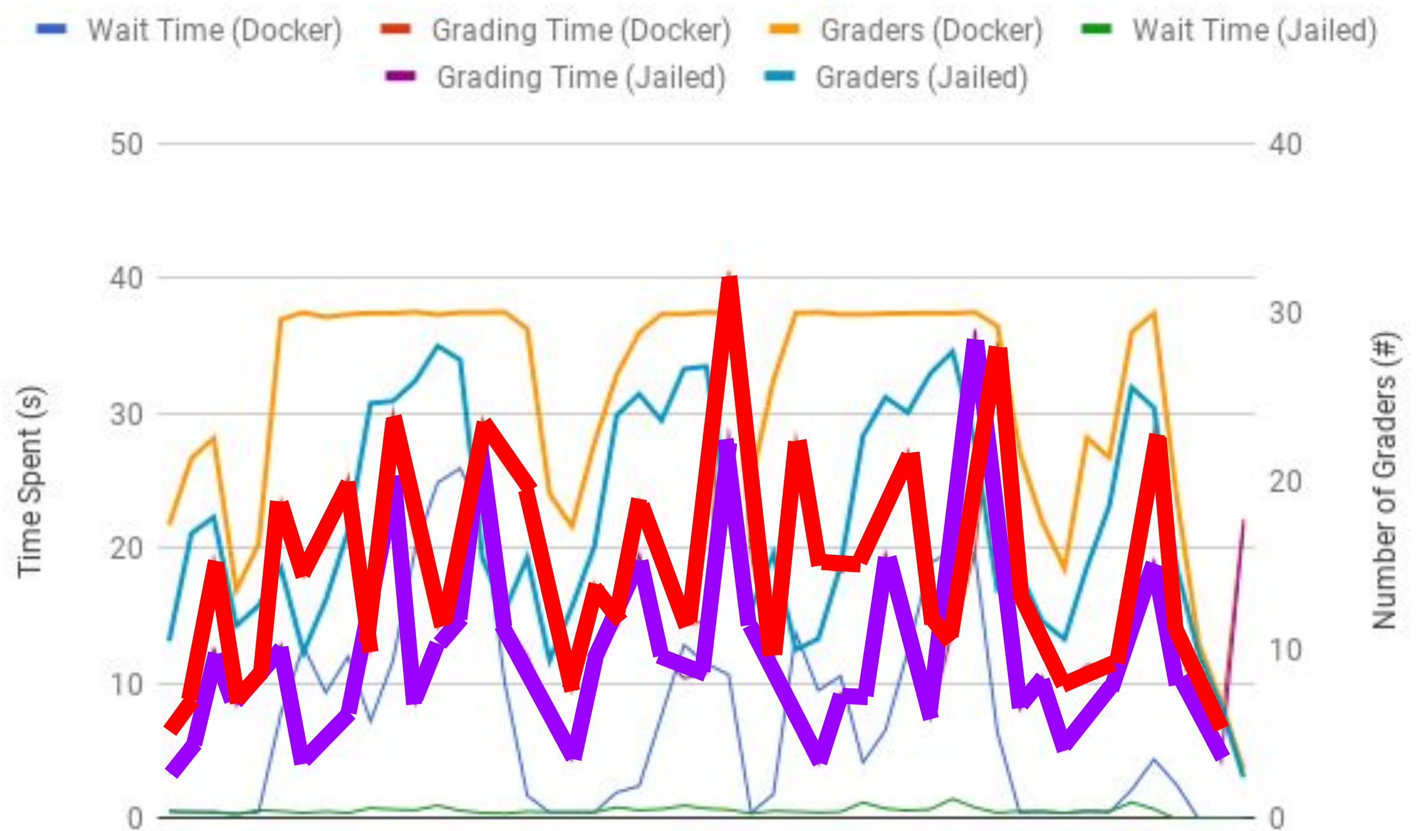


Increase of RAM for Docker (peak 15%)

Results (16x) - Time Spent

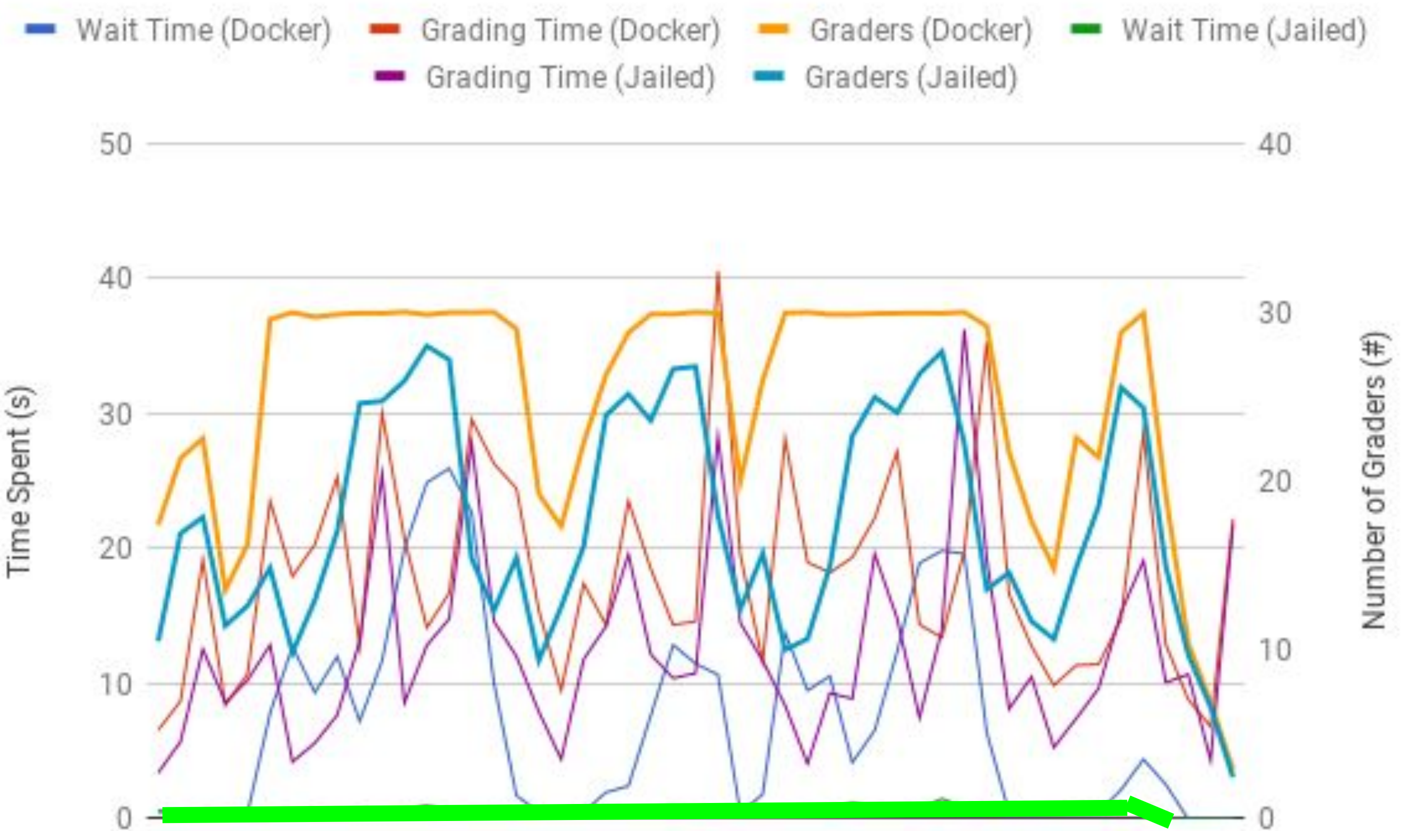


Results (16x) - Time Spent



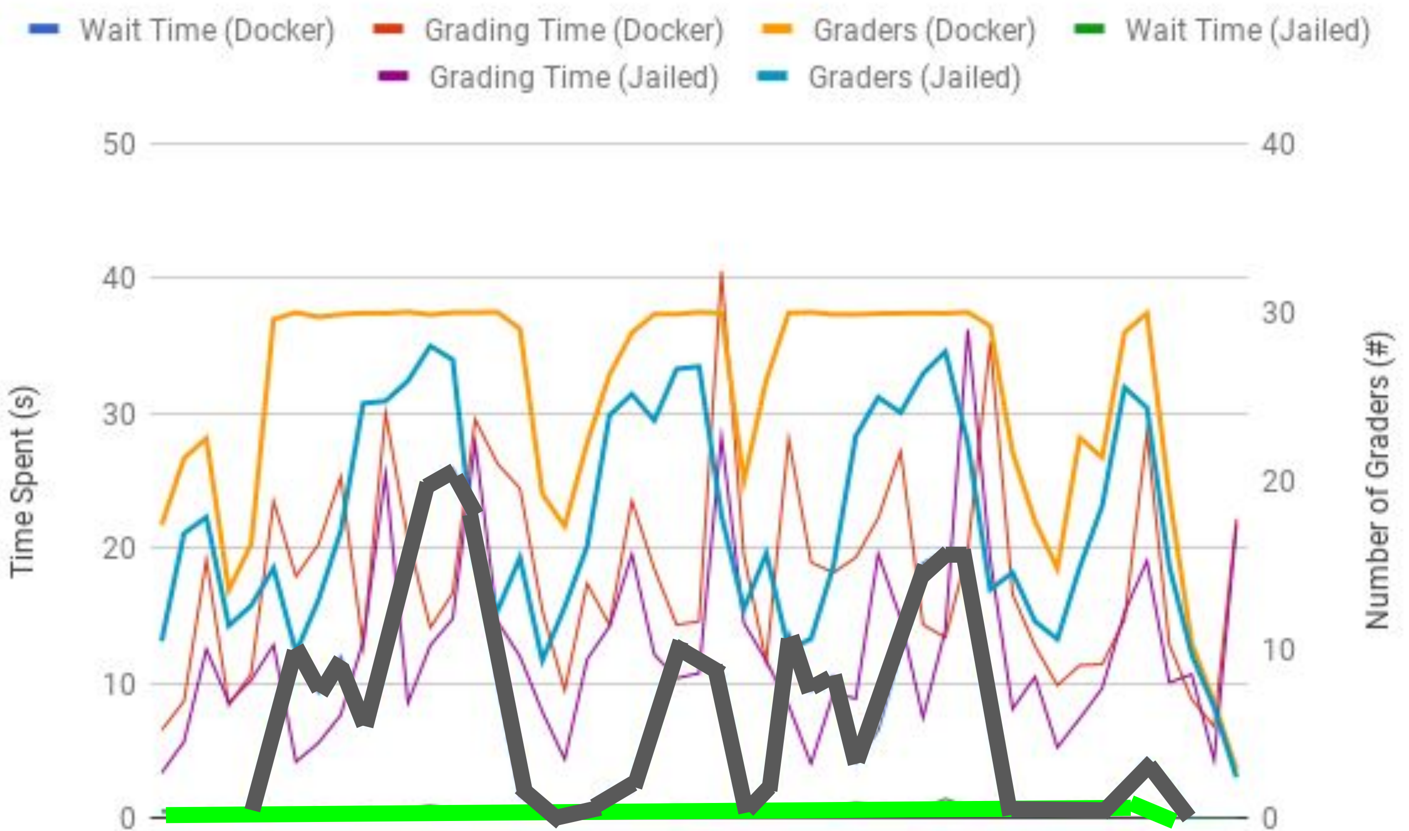
There are spikes in the grade time taken for Docker (Red) vs Sandbox (Purple)

Results (16x) - Time Spent



Wait time for Jailed is ~0s

Results (16x) - Time Spent



Wait time for Docker spikes while jailed remains constant

Takeaways

- When not stressing all available graders and <10 concurrent containers:
  - Docker and Sandbox used similar amounts of CPU and RAM and had similar grading throughputs
  - Docker increased times on average by ~2.5s per submission:
    - It took ~*1.5s* to create the containers
    - It took ~*1s* to destroy the containers
- When Docker used all available graders and had >30 concurrent containers:
  - Docker used on average more CPU (peak *5%*) and RAM (peak *15%* more)
  - The time to handle Docker related tasks increased, causing drop in throughput
  - It took ~*7.5s* to create the containers
  - It took ~*3.3s* to destroy the containers
  - Largest measured peak was ~*25s* additional time, but average of ~*10s* throughout the hour
- **Sandbox is more efficient in the extreme case, but not really for normal/expected workloads, but Docker solves the language/version problem**

Future Work

- Look into scheduling algorithms for pre-spinning and destroying containers
  - Create pool of live containers to be used for grading
  - Have to base what containers are in pool based on previous submissions over some time slice *T*
  - Pool should tune and optimize itself automatically as submissions come in
  - Look into separating out container destruction to separate reaper process
  - Cannot allow too many concurrent containers to be alive at one time as potential for making system unstable
- Create web-based toolchain for instructors to easily create Dockerfiles for their gradeables

Rensselaer

Submitty

Rensselaer Center for Open Source

SCAN ME

https://submitty.org/

For More Submitty at SIGCSE:

Go Back in Time To Previous Paper:
Autograding Distributed Algorithms in Networked Containers

Tomorrow @3pm, Poster:
Lichen: Customizable, Open Source Plagiarism Detection in Submitty

Tomorrow @3pm, Poster:
Facilitating Discussion-Based Grading And Private Channels via an Integrated Forum