Automated Benchmarking of Container Applications

Paulius Dilkas

18th July 2019

1 Architecture and Implementation

Two simple Dockerfiles were created. The first one is used for TaskManager and JobManager pods and extends the original Flink Docker image by enabling Prometheus support on port 9250. Prometheus can then query that port in order to retrieve and record performance metrics.

The second Dockerfile extends the first one by adding a JAR file with Java code for both the control server and the Flink app. This image also contains a custom `ENTRYPOINT` that sends the Flink app to the JobManager (via port 8081) as a background process, while executing the control server in the foreground. This is the optimal arrangement of the two processes since the control server always waits for the Flink app to finish in order to take its running time into account when requesting data from Prometheus.

In both cases, one needs to change the permissions and group ownership of the `/opt/flink` directory (to `775` and `root`, respectively) so that the containers can be successfully executed by any user belonging to the group `root`. Both images were put on Docker Hub, as this was the simplest way to make them available to MiniShift.

A Docker Compose file was written with three services: Control, JobManager, and TaskManager, establishing open ports as pictured in Figure 3). This file was then converted to OpenShift manifestos using Kompose[1]. The generated manifestos, relevant network connections, and other dependencies are displayed in Figure 3. The entire system can then be updated and deployed by generating a new JAR file using Maven, building and uploading the two Dockerfiles, and recreating all components of the OpenShift configuration, as described by the manifestos.

Prometheus add-on for MiniShift[2] was modified to disable OAuth-based authentication by replacing `-skip-auth-regex=^/metrics` with `-skip-auth-regex=^/`. Furthermore, its configuration file was updated to set both scrape and evaluation intervals to 1 second and the list of targets to JobManager and TaskManager, both on port 9250.

Configuration Files component represents a ConfigMap created using the `oc` command that contains two configuration files, `global.yaml` and `components.yaml` (see Figures 1 and 2 for examples). The former contains basic networking information along with two parameters that control the experiment (`experimentLength` and `messagesPerSecond`) and a list of metrics. The two experiment-specific parameters together define the frequency and number of messages that the `ControlServer` will send to the `Benchmarker`. Each metric is described with three properties: `name`, `filename`, and `query`. The last one corresponds to the name of the property as defined by Prometheus, while the other two are used for data storage and plotting. The `components.yaml` configuration file, on the other hand describes a sequence of processing stages, each with its own CPU usage time, memory usage, and output data size (i.e., the amount of data passed to the next stage).

We illustrate some aspects of the execution and how different components communicate with each other in Figure 4. After the Flink app (called Benchmarker) is initialised, it immediately establishes the control server as a `socketTextStream`, i.e., the initial source of data. It then constructs a chain of mappers according to the `components.yaml` specifications.

[1] `http://kompose.io/`

[2] `https://github.com/minishift/minishift-addons/tree/master/add-ons/prometheus`

```yaml
controlHostname: control
controlPort: 9998
prometheusHostname: prometheus
messagesPerSecond: 1
experimentLength: 3 # in seconds
metrics:
  - name: Throughput
    filename: throughput
    query: flink_taskmanager_job_task_operator_componentThroughput
  - name: Heap Usage (MiB)
    filename: heap
    query: flink_taskmanager_Status_JVM_Memory_Heap_Used
  - name: CPU Load
    filename: cpu
    query: flink_taskmanager_Status_JVM_CPU_Load
```

Figure 1: Example global configuration file

```yaml
- cpuTime: 5000 # in ms
  memoryUsage: 100 # in MiB
  outputSize: 1 # in KiB
- cpuTime: 5000
  memoryUsage: 200
  outputSize: 1
```

Figure 2: Example configuration file that defines a list of components with their resource requirements

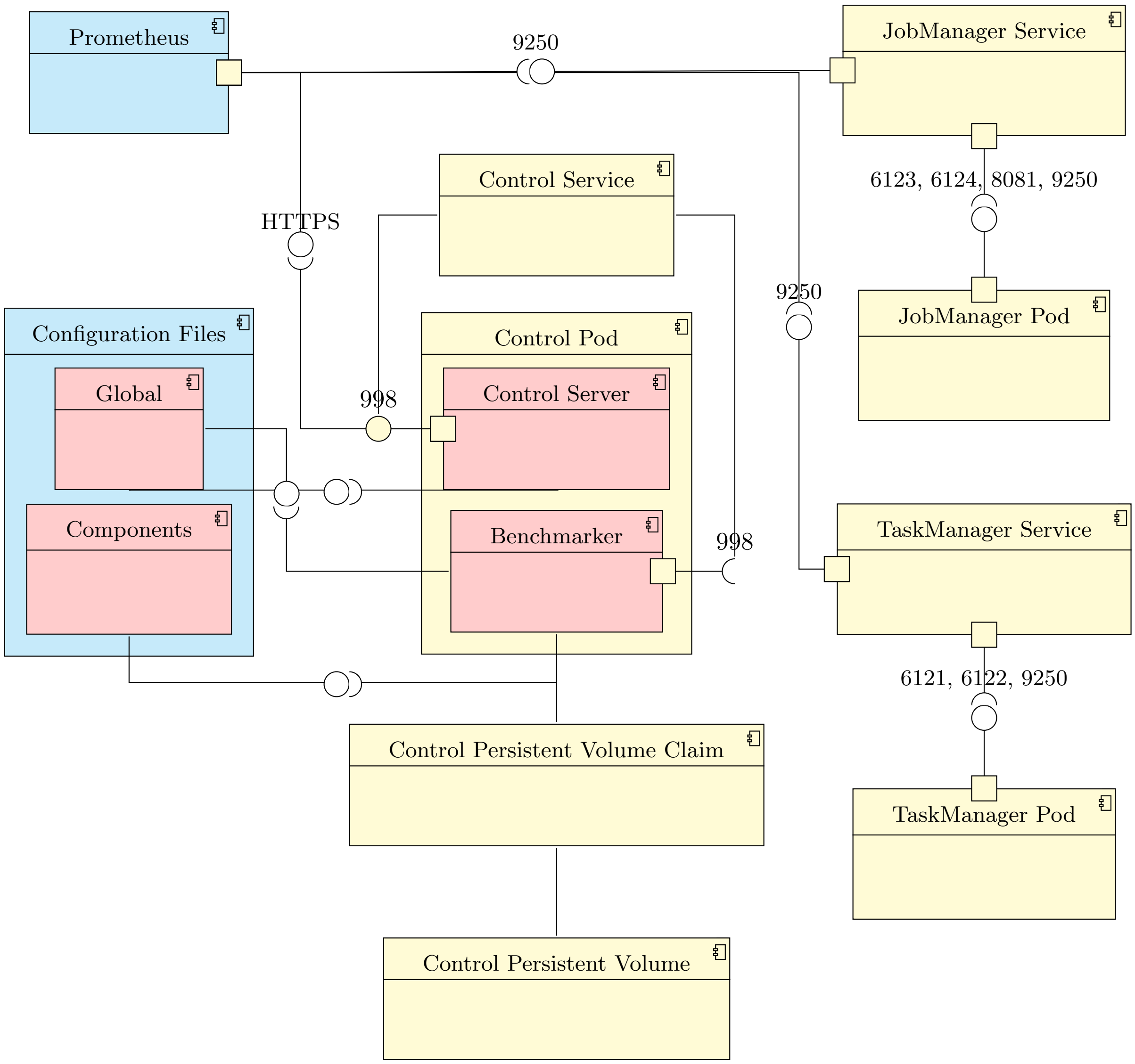


Figure 3: UML component diagram of the system, as deployed on MiniShift. Yellow components are OpenShift manifestos, while red components represent files (either Java classes or YAML configuration files). Network connections are shown with ports and have port numbers (or application-layer protocol names) displayed.

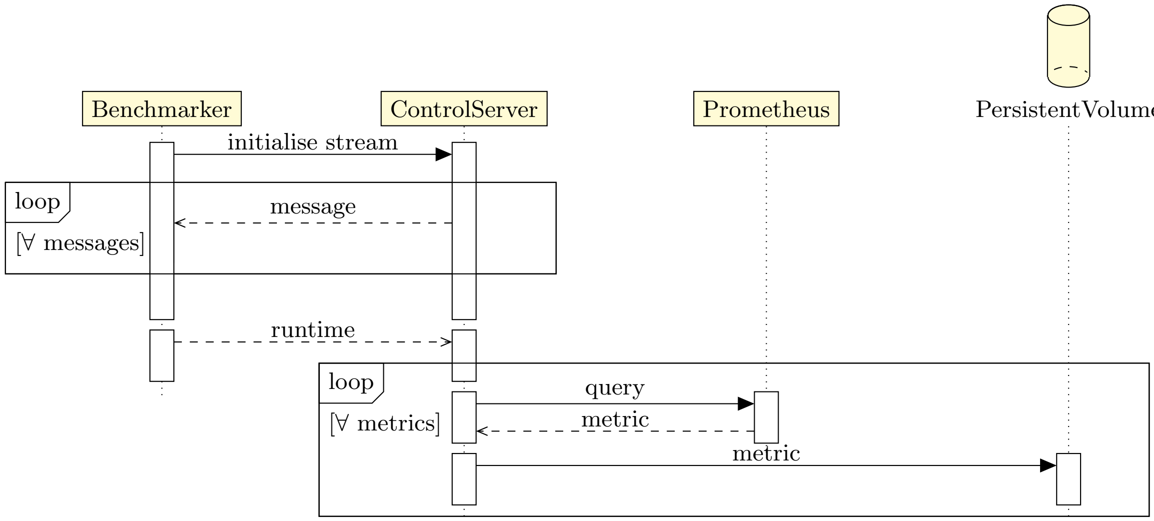


Figure 4: Communication between different parts of the system visualised as a UML sequence diagram

The control server periodically sends messages to Benchmarker (as defined in `global.yaml`). Each component (mapper) does three things upon receiving each message:

1. First, it allocates an array so that the total memory usage would be as close to `memoryUsage` as possible. The array size is calculated using a linear regression model established using experimental data (see Section 2).

2. Then, it creates a `String` object taking up `outputSize` KiB of memory. This string will be passed to the next component in the chain.

3. Finally, it spends the remaining time (until total execution time is exactly `cpuTime`) testing the Collatz conjecture [1] one initial integer at a time.

After all messages from the control server pass through every mapper, Benchmarker connects to the control server, sending it total running time (as measured by `JobExecutionResult.getNetRuntime()`). This number is then rounded up to an integer number of minutes and used to retrieve performance data for the time interval when the application was running.

Finally, for each metric defined in the global configuration file, the control server establishes an HTTPS connection to Prometheus, collects JSON data recording the values of that metric in the last few minutes (as calculated previously), and writes the data to a file (separate for each metric) on the persistent volume.

The last component of the system consists of two Python scripts. The first is responsible for deploying the entire OpenShift setup, waiting for the control server to terminate, and using MiniShift SSH to copy files from the persistent volume to a host folder, which places them into a local directory on the host machine. The second script can then read the data and plot it using Matplotlib, comparing observed data with their expected values, as dictated by the parameters of the experiment.

2 Local Performance Tuning

The component script, responsible for using predefined amounts of resources, was tested and adjusted locally, ensuring that it uses 100% of a single CPU and `memoryUsage` MiB of memory. Total heap memory usage was measured for array sizes $2^0, 2^1, 2^2, \ldots, 2^9$ and output strings of $2^0, 2^1, 2^2, \ldots, \min\{2^8, \text{array size}\}$ characters (the output string is constructed using the array, so the array size must always be at least as big as the

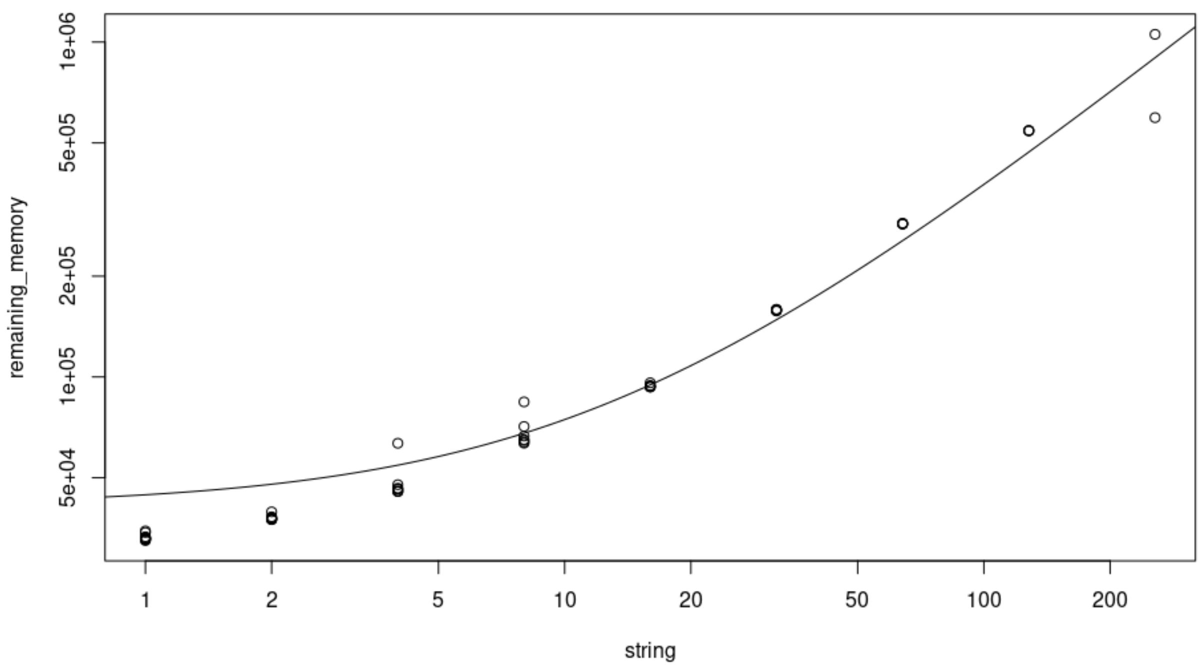


Figure 5: A log-log plot comparing the number of characters in a string and the observed memory unaccounted by the array. In each column, separate points correspond to different array sizes. The curve is a best-fit linear regression line.

output string). Maximum heap usage was measured using GNU Time[3] and its Maximum Resident Set Size metric. Each experiment was repeated three times, and median values were taken.

We seek to know the average amount of memory used by a single character of a Java string. Knowing that a byte on an array takes up exactly one byte allows us to reformulate the problem to a simple linear regression shown in Figure 5. The model shows that overall memory usage can be expressed as

$$\text{memory usage} = 40\,\text{MiB} + \text{array size} + 3.268 \times \text{string size} + \epsilon, \tag{1}$$

contradicting the common wisdom that a character uses approximately two bytes of memory [2].

Figure 6 presents a more detailed view, however suggesting the same conclusion. While the predictions seem to consistently overestimate memory consumption for short strings and similarly underestimate it for longer strings, adding a quadratic term is not enough to remove the bias in errors, and the errors are sufficiently small (see Section 2.1 for more details).

2.1 Adjusted Performance

We can use the two numerical parameters in Equation (1) to adjust our 'mapping function' in order to ensure that it uses the correct amount of memory. We run a similar set of experiments as before, except replacing array size with expected memory usage as one of our independent variables (the other being string size). Memory usage is set to four different values: 64, 128, 256, and 512 MiB (note that the smallest possible memory usage is about 40 MiB), while string size is exponentially increased from 1 MiB up to the largest

[3] `https://www.gnu.org/software/time/`

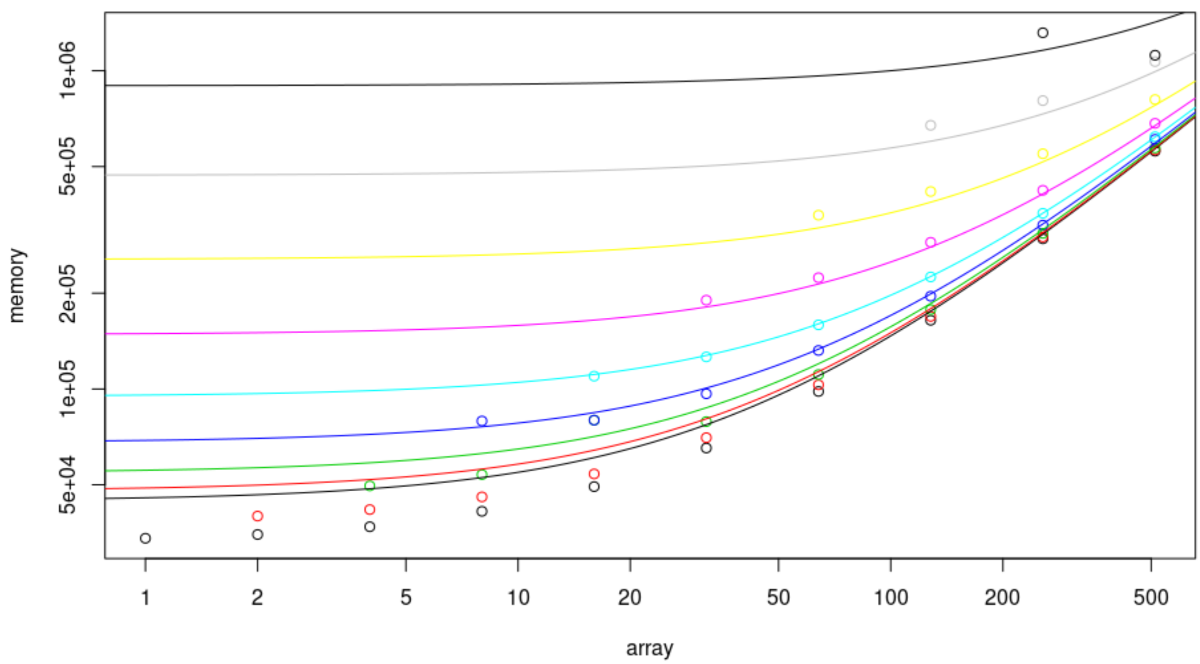


Figure 6: A log-log plot showing memory consumption across a range of array sizes, with different string sizes represented by different colours. For each string size, we also draw a regression line in the corresponding colour.

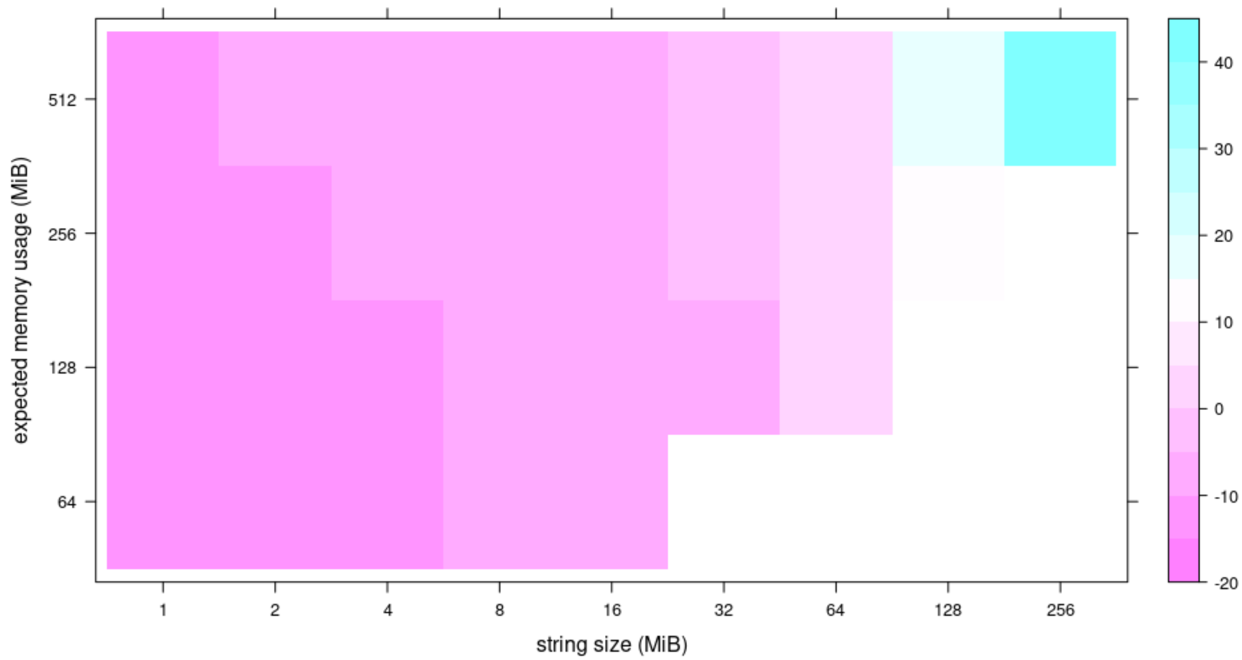


Figure 7: A heat map of errors (in KiB)

power of two small enough so that the string can be constructed from the array. We plot the errors in Figure 7. Note that the largest error is smaller than 50 KiB, which is good enough for our needs.

3 Experimental Evaluation

Experiments were performed in order to determine how well performance metrics observed with a standalone Java application transfer to MiniShift. We explore four values of `memoryUsage` (64 MiB, 128 MiB, 256 MiB, 512 MiB), while keeping `cpuTime` at 0 so that each run lasts only as long as it takes to allocate and randomise the memory. For `outputSize`, we explore every power-of-two number of MiB compatible with the current `memoryUsage` value. We stick to a single component and record CPU and memory consumption at 1 sec intervals using Prometheus. Each `memoryUsage` and `outputSize` configuration is written into `components.yaml` and run three times. With each run, we recreate all OpenShift components (pods, services, etc.), wait for the control server to terminate, and retrieve the generated JSON files.

Figure 8 shows CPU usage across all runs. Even though our standalone Java application easily reaches 100% CPU usage, when transferred to an OpenShift environment, a typical run could only get around 10%–15% (as indicated by the red curve), occasionally reaching up to 70% or 80% CPU usage. This can be explained by the fact that MiniShift internal processes as well as Flink JobManager and TaskManager are all running on the same machine. Even though the processes are distributed among eight cores, this overhead is sufficient to significantly decelerate the application.

Figure 9 shows similar memory usage measurements divided into four plots, one for each value of `memoryUsage`. We can see that there is significant variation among runs (and different `outputSize` values). In fact, in order to determine whether memory usage is optimal or hampered, one would need to run many identical experiments to account for variability. Moreover, each curve is unlikely to be fully summar-

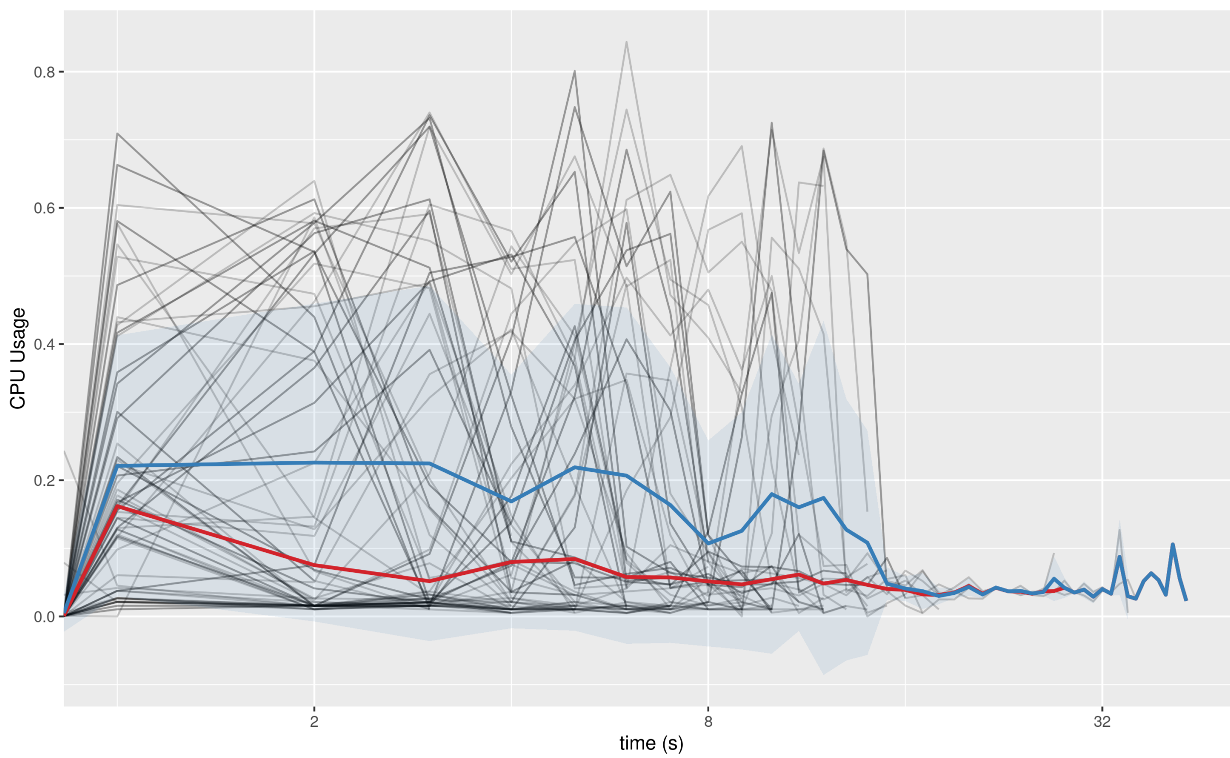
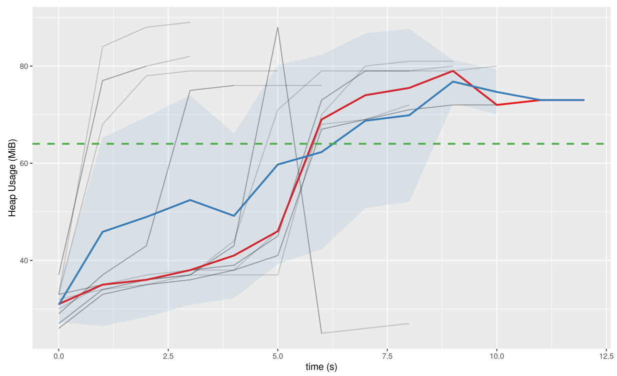
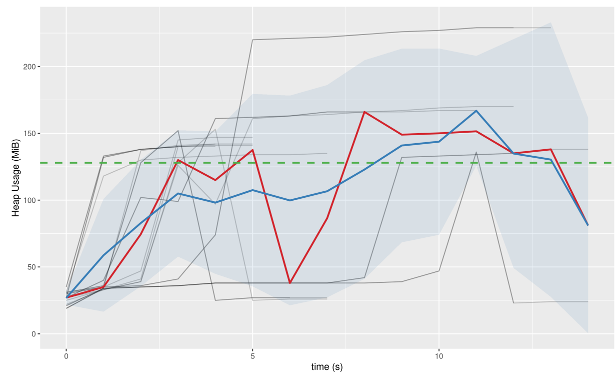


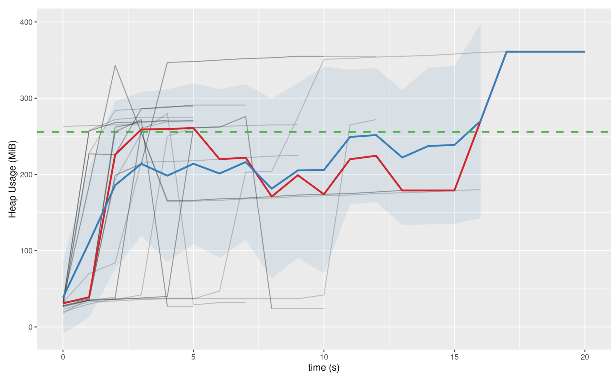
Figure 8: CPU usage across time. Each gray line represents a different run. The red curve is their (pointwise) median, the blue curve is the mean, while the shaded area marks one standard deviation around the mean. Note that time is on a log scale.



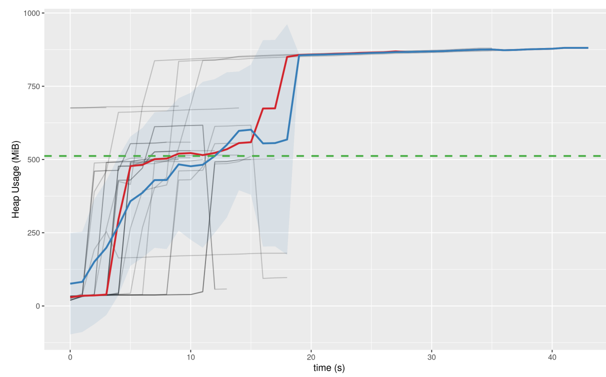
(a) expected heap usage: 64 MiB



(b) expected heap usage: 128 MiB



(c) expected heap usage: 256 MiB



(d) expected heap usage: 512 MiB

Figure 9: Observed versus expected memory usage across time for four expected memory amounts. The green dashed horizontal line marks the expected amount of memory usage. Each gray line represents a different run. The red curve is their (pointwise) median, the blue curve is the mean, while the shaded area marks one standard deviation around the mean.

ised by a single number: maximum values are almost always higher than the expected result, while means are likely to be distorted by the initial several seconds of low memory usage as well as observed dips in memory usage later in the execution.

Note that individual runs can be summarised as follows:

1. Memory usage starts low.
2. It rises two times.
3. Sometimes memory usage experiences a significant drop, and sometimes this step is skipped.
4. Memory usage stays constant for a while.
5. The process terminates.

We can easily explain this pattern. The first increase is caused by the array allocation, while the second one is the result of constructing the output string. The drop in memory usage happens when the array is deallocated (garbage-collected) some time after the execution of my code completes. Sometimes that happens early enough to be captured by Prometheus, and sometimes the Flink job is marked as complete before garbage collection activates.

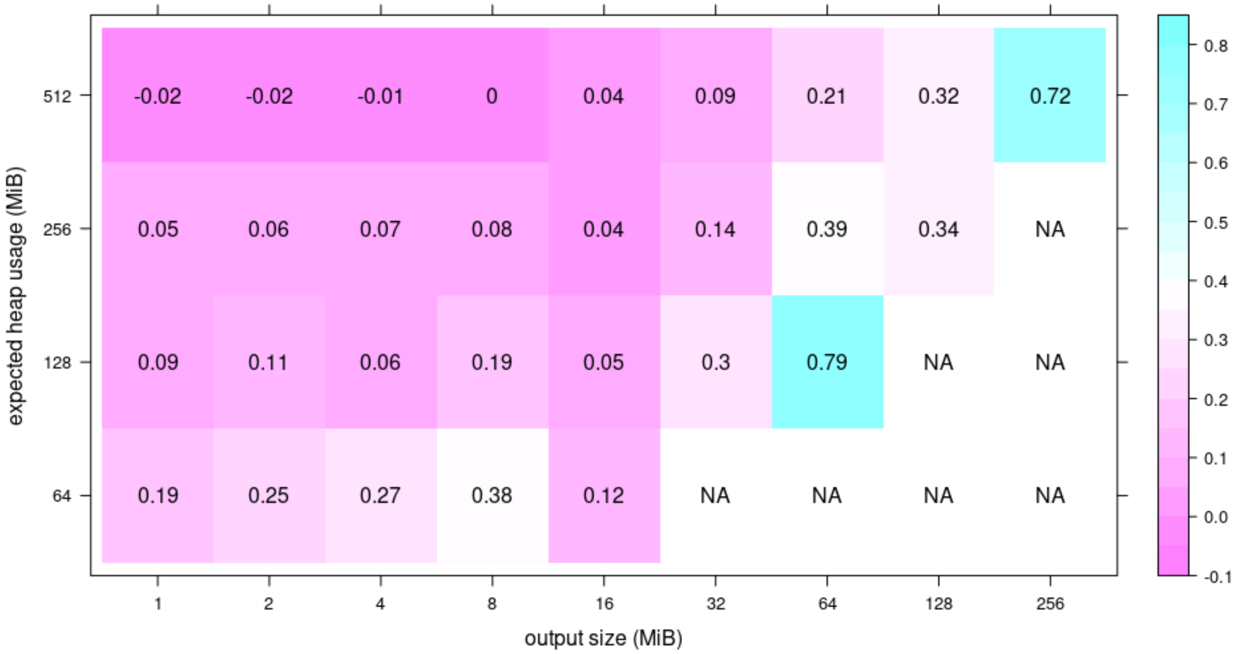


Figure 10: Relative median memory usage errors, as predicted by the maximum memory usage across execution

Finally, we consider the extent to which memory usage can be summarised by taking the maximum across time. We report each difference between expected E and observed O values as a relative error, i.e.,

$$\frac{O - E}{E}.$$

We consider these errors for all viable combinations of `memoryUsage` and `outputSize` and report the median of the three identical runs performed on each combination. The results are in Figure 10. Unsurprisingly, maximum memory usage across time is usually higher than the estimate. Also note that the overall shape of the heat map is similar to Figure 7, where we measure differences between observed and expected memory usage with the standalone Java application. In both cases, observed values are smaller with lower values of `outputSize`, and a combination of high overall memory usage and a long output string in the top right corner of both heat maps is likely to result in observed memory usage being significantly higher than the expected value. Even though we take median values to reduce the effect of outliers, observed memory usage can be up to 80% higher than the expected value, adding evidence to the imprecision and unreliability of making judgments based on a single number or a single experiment.

References

[1] J. J. O'Connor and E. F. Robertson. Lothar Collatz. `http://www-history.mcs.st-andrews.ac.uk/Biographies/Collatz.html`, November 2006.

[2] M. Vorontsov. An overview of memory saving techniques in Java. `http://java-performance.info/overview-of-memory-saving-techniques-java/`, June 2013.